HMPI: A Message-Passing Library for Heterogeneous Networks of Computers

Manumachu Ravi Reddy

Doctor of Philosophy Dissertation

June 2005

Head of Department: Barry Smyth
Supervisor: Alexey Lastovetsky

Computer Science Department,
University College Dublin,
Belfield,
Dublin.

To my parents…

Manumachu Sree Rami Reddy

Manumachu Koteswari

Acknowledgements

During the course of this research, I have had great privilege to work with a very competent group of people. I would like to take this opportunity to thank every one of them.

I would like to thank my supervisor, Alexey Lastovetsky, for being available at all times to offer valuable advice and guidance. I'm very happy to have continued my collaboration with Alexey from industry into academia. We have had many insightful discussions, which have greatly helped me getting to understand the nitty-gritty of the area of heterogeneous computing. I'm grateful to Alexey Kalinov and Neil Hurley for providing valuable feedback on my dissertation. I would like to acknowledge the useful review comments provided by Jack Dongarra on the design of the interfaces in my research work.

I'm greatly indebted to the staff and postgraduates of the computer science department. I'm very grateful to Gerry Dunnion for providing and administering the computing resources to run my parallel experiments. I appreciate all the assistance provided by Patricia Geoghegan in dealing with the administrative formalities. I would like to thank the members of the postgraduate soccer team for all the laughs on and off the soccer pitch.

I would like to thank my friends -- Abhay, Leena, Bart, Narinder, Pramilla, Victor, Ajit, Rahul, Pracchi, Anirudh, Andy, Viv and Jeff -- for keeping me motivated through this arduous but exciting journey. I would specially like to thank Srinivas Chillara for taking time out to proofread and review this dissertation and Hanumantha Rao for giving words of wisdom related to the research experience.

And finally, I would like to thank my parents, who throughout my life have always been there to give me encouragement and support.

Table of Contents

Table of Figures

List of Tables

Relevant Publications

A. Lastovetsky and R. Reddy, “HMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers,” To appear in Journal of Parallel and Distributed Computing (JPDC), 2005.

A. Lastovetsky and R. Reddy, “Data Partitioning for Multiprocessors with Memory Heterogeneity and Memory Constraints,” To appear in Special Issue of Scientific Programming, 2005.

A. Lastovetsky and R. Reddy, “A Variable Group Block Distribution Strategy for Dense Factorizations on Networks of Heterogeneous Computers,” in Proceedings of the 6th International Conference on Parallel Processing and Applied Mathematics, Workshop on HPC Linear Algebra Libraries for Computers with Multilevel Memories, October 2005.

A. Lastovetsky and R. Reddy, “Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers,” Submitted to International Journal of High Performance Computations and Applications (IJHPCA).

A. Lastovetsky and R. Reddy, “Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers with Task Size Limits,” In Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks

(ISPDC/HeteroPar'04), IEEE Computer Society Press, pp.133-140, 5-7 July 2004, Cork, Ireland, 2004.

A. Lastovetsky and R. Reddy, “Data Partitioning with a Realistic Performance Model of Networks of Heterogeneous Computers,” In Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), 26-30 April 2004, Santa Fe, New Mexico, USA, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2004.

A. Lastovetsky and R. Reddy, “On Performance Analysis of Heterogeneous Parallel Algorithms,” In Parallel Computing, Volume 30, No. 11, pp.1195-1216, 2004.

A. Lastovetsky and R. Reddy, “Classification of Partitioning Problems for Networks of Heterogeneous Computers,” In Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics (PPAM 2003), Czestochowa, Poland, Lecture Notes in Computer Science, 3019, pp.921-929, 2003.

A. Lastovetsky and R. Reddy, “An Approach to Assessment of Heterogeneous Parallel Algorithms,” In the 7th International Conference on Parallel Computing Technologies (PaCT 2003), Nizhni Novgorod, Russia, Lecture Notes in Computer Science, 2763, pp. 117-129, 2003.

A. Lastovetsky and R. Reddy, “HMPI: Towards a Message-Passing Library for Heterogeneous Networks of Computers,” In Proceedings of the 17th International Parallel and Distributed

Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2003.

Abstract

Heterogeneous networks of computers (HNOCs) are becoming an increasingly popular platform for executing parallel applications. HNOCs have been proposed as a viable and cost-effective alternative to supercomputers because of the computation power they offer. The use of such networks for parallel high-performance computing is limited only by the absence of appropriate system software.

We propose Heterogeneous Message Passing Interface (HMPI), a unifying framework designed specially for programming high-performance computations on HNOCs. HMPI provides all the features to the user to write portable and efficient parallel applications on HNOCs. These features automate all the main stages involved in application development on HNOCs:

Stage-1). Determination of the characterization parameters relevant to the computational requirements of the applications and the machine capabilities of the heterogeneous system. These are mainly the speeds of processors, the latencies and bandwidths of the communication links between them, and the user-available memory capacity of each machine. The largest problem size that can be run is limited by the user-available memory capacity on a given machine. These parameters are determined before the application execution and form the model of executing network of computers. HMPI provides interfaces to update the parameters of the model at runtime taking into account the fluctuations of the network load.

Stage-2). Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes. This step of heterogeneous decomposition is parameterized by the characterization parameters determined in the first step. We propose Heterogeneous Data Partitioning Interface (HDPI), which automates this step of heterogeneous decomposition. HDPI provides API that allows the application programmers to specify simple and basic partitioning

criteria in the form of parameters and functions to partition the mathematical objects used in their parallel applications.

Stage-3). Selection of the optimal set of processes running on different computers of the heterogeneous network by taking into account the speeds of the processors, and the latencies and the bandwidths of the communications links between them. HMPI provides a small set of extensions to MPI, which automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group. The main goal of the design of this API in HMPI is to smoothly and naturally extend the MPI model for HNOCs.

Stage-4). Application program execution on the HNOCs. The command line user interface of HMPI developed consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of the HMPI application programs on the virtual parallel machine. The notion of virtual parallel machine enables a collection of heterogeneous computers to be used as single large parallel computer.

The merits of HMPI are demonstrated through the design, analysis, and implementation of three applications on HNOCs. They are Matrix-matrix multiplication, Cholesky Factorization, and EM3D. These applications are representative of many scientific applications. Experimental results show that carefully designed HMPI applications can show very good improvements in execution performance on HNOCs.

Once developed, an HMPI application will run efficiently on any HNOCs without any changes to its source code (we call the property *efficient portability*). It can be seen that the improved performance of the HMPI applications is not due to the fine-tuning of these applications to a specific environment. By hiding the non-uniformity of the underlying heterogeneous system from the application programmer, the HMPI offers an environment that encourages the design of heterogeneous parallel software in an architecture-independent manner.

CHAPTER 1

Introduction

Heterogeneous networks of computers (HNOCs) are becoming an increasingly popular platform for executing parallel applications [ACP95, SDA96]. HNOCs have been proposed as a viable and cost-effective alternative to supercomputers because of the computation power they offer. The use of such networks for parallel high-performance computing is limited only by the absence of appropriate system software.

The standard Message Passing Interface (MPI) [SOJ+96] is the main programming tool used for programming high-performance computations on homogeneous distributed-memory computer systems such as supercomputers and clusters of workstations. It is also normally used to write parallel programs for heterogeneous networks of computers (HNOCs). However, it does not provide tools that address some additional challenges posed by HNOCs, which are outlined below:

- **Heterogeneity of processors**. A good parallel application for HNOCs must distribute computations unevenly taking into account the speeds of the processors. The efficiency of the parallel application also depends on the accuracy of estimation of the speeds of the processors of the HNOCs, which is difficult because the processors may demonstrate different speeds for different applications due to differences in the set of instructions, the number of instruction execution units, the number of registers, the structure of memory hierarchy and so on.
- **Ad hoc communication network**. The common communication network is normally heterogeneous. The latency and bandwidth of communication links between different pairs of processors may differ significantly. This makes the problem of optimal

distribution of computations and communications across the HNOCs much more difficult than across a dedicated cluster of workstations interconnected with a homogeneous high-performance communication network.

- **Multi-user decentralized computer system**. Unlike dedicated clusters and supercomputers, HNOCs are not strongly centralized computer systems. A typical HNOC consists of relatively autonomous computers, where each one may be used and administered independently by its users. The first implication with the multi-user decentralized nature of HNOCs is the unstable performance characteristics of processors during the execution of a parallel program as the computers may be used for other computations and communications.

Thus the standard MPI does not provide features, which facilitate the writing of parallel programs that distribute computations and communications unevenly, taking into account the speeds of the processors, and the latencies and bandwidths of communication links. To the best of our knowledge, there is no research effort made to address this challenge. We present an effort in this direction – a small set of extensions to MPI, called HMPI (Heterogeneous MPI), aimed at efficient parallel computing on HNOCs, and its research implementation. The main goal of the design of the API in HMPI is to smoothly and naturally extend the MPI model for heterogeneous networks of computers. This involves the design of a layer above MPI that does not involve any changes to the existing MPI API. The HMPI API must be easy-to-use and suitable for most scientific applications. The HMPI API must also facilitate transformation of MPI applications to HMPI applications that run efficiently on HNOCs.

Application development on HNOCs consists of four stages:

Stage-1). Determination of characterization parameters relevant to both the computational requirements of the applications and the machine capabilities of the heterogeneous system using information about the expected types of application problems and the machines in the heterogeneous system. For example, for floating-point operations, the computational requirements to be quantified are the number of each type of floating-point operation needed to perform the calculation, and the capabilities of the machines to be quantified are the speeds for these different types of floating-point operations, the latencies and the bandwidths of the communication links between each pair of machines, and the user-available memory capacity of each machine. The largest problem size that can be run is limited by the user-available memory capacity on a given machine. Other parameters such as the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy on each machine, memory latency associated with each level of memory hierarchy, multiple instruction issue, instruction pipelining etc are incorporated into the notion of speed of a processor.

Stage-2). Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes. This step of heterogeneous decomposition is parameterized by the characterization parameters determined in the first step, mainly, the speeds of processors and the latencies and bandwidths of the communication links between them, and the user-available memory capacity of the machine.

Stage-3). Matching and scheduling (mapping). The information generated in the previous stages are used to derive the estimated execution time for a given sub-problem on a given machine and the intermachine communication overhead associated with a given assignment of sub-problems to machines. These static results and dynamic information about the current load and status of the interconnection network can be used to determine the assignment of the sub-problems to parallel processes and the mapping of these parallel processes to the computers of the executing network, that is, the selection of an optimal set of processes running on different computers of the heterogeneous network by taking into account the speeds of the processors, and the latencies and the bandwidths of the communications links between them.

Stage-4). Application program execution on the HNOCs.

The first stage in the development of HMPI involved the automation of steps 1, 3, and 4. It involved the design of a small set of extensions to MPI that can be used for

- Determination of the characterization parameters relevant to the computational requirements of the applications and the machine capabilities of the heterogeneous system, and
- Selection of the optimal set of processes running on different computers of the heterogeneous network.

This was followed by an implementation of these set of extensions. The command line user interface of HMPI developed consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of the HMPI application programs on the virtual parallel machine. The notion of virtual parallel machine enables a collection of heterogeneous computers to be used as single large parallel computer. Rather than leaving the parallel programmer to manually select each individual computer where tasks are to execute and then log into each machine in turn to actually spawn the tasks and monitor the execution, the virtual machine provides a simple abstraction to encompass the disparate machines.

While using HMPI for parallel solution of regular and irregular problems on HNOCs, we found that the second step of heterogeneous decomposition can be very tedious and error-prone.

An *irregular* problem is characterized by some inherent coarse-grained or large-grained structure. This structure implies a quite deterministic decomposition of the whole problem into relatively small number of subtasks, which are of different size and can be solved in parallel. Correspondingly, a natural way of decomposition of the whole program, which solves the irregular problem on a network of computers, is a set of parallel processes, each solving its subtask and all together interacting via message passing. As sizes of these subtasks are typically different, the processes perform different volumes of computation. Therefore, the mapping of these processes to the computers of the executing HNOC should be performed very carefully to ensure the best execution time of the program.

The most natural decomposition of a *regular* problem is a large number of small identical subtasks that can be solved in parallel. As those subtasks are identical, they are all of the same size. Multiplication of two $n \times n$ dense matrices is an example of a regular problem. This problem is naturally decomposed into n^2 identical subtasks, each of which is to compute one element of the resulting matrix. The main idea behind an efficient solution to a regular problem on a heterogeneous network of computers is to transform the problem into an irregular problem, the structure of which is determined by the structure of the executing network rather than the structure of the problem itself. So, the whole regular problem is decomposed into a set of relatively large sub-problems, each made of a number of small identical subtasks stuck together. The size of each subproblem, that is, the number of elementary identical subtasks constituting the subproblem, depends on the speed of the processor, on which the subproblem will be solved. Correspondingly, the parallel program, which solves the problem on the heterogeneous network of computers, is a set of parallel processes, each solving one subproblem on a separate physical processor and all together interacting via message passing. The volume of computations performed by each of these processes should be proportional to its speed.

Thus while the step of problem decomposition is trivial for irregular problems, it becomes key for a regular problem. In fact, at this very step the application programmer designs a heterogeneous data parallel algorithm by working out a generic decomposition of the regular problem parameterized by the number and speed of processors. Most typically the generic decomposition takes the form of data partitioning. The application programmers need to solve corresponding data partitioning problems and design and implement all supportive code from scratch. Existing programming systems for heterogeneous parallel computing [AKL+99, LAK+00, Las02] support the mapping of parallel algorithms to the executing network but provide very poor support for generic heterogeneous decomposition of regular problems parameterized by the number and speed of processors. This motivated us to try and automate the step of heterogeneous decomposition of regular problems by designing a library of functions solving typical partitioning problems for networks of heterogeneous computers. Our original approach was to do it by just collecting existing algorithms, designing an API to these algorithms and implementing the API. The main problem we came across was that no classification of partitioning problems was found which could be used as a basis of API design. Existing algorithms created a very fragmented picture. Therefore an important goal of our research became to classify partitioning problems for networks of heterogeneous computers. Such classification had to help to specify problems with known efficient solutions and identify open problems. Then based on this classification an API would have to be designed and partially implemented (for problems that have known efficient solutions). An additional requirement to this classification was that it had to be useful for distributed computing on networks as well.

Therefore, we propose a classification of mathematical problems encountered during partitioning of data when designing parallel algorithms on networks of heterogeneous computers. Our approach to this classification is based on two corner stones:

- A realistic performance model of networks of heterogeneous computers,
- A natural classification of mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) problem solving on networks of heterogeneous computers.

Our classification of problems consists of two categories: problems with known efficient solutions and open problems. Based on this classification, we suggest an API for partitioning mathematical objects commonly used in scientific and engineering domains for solving problems on networks of heterogeneous computers. The API is part of the Heterogeneous Data Partitioning Interface (HDPI). These interfaces allow the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition their mathematical objects. These partitioning interfaces are designed to be used along with various programming tools for parallel and distributed computing on heterogeneous networks.

We evaluate HMPI using three applications on HNOCs. They are Matrix-matrix multiplication, Cholesky Factorization, and the EM3D application simulating the interaction of electric and magnetic fields on a three-dimensional object. These applications are representative of many scientific applications. Experimental results show that carefully designed HMPI applications can show very good improvements in execution performance on HNOCs.

The ultimate goal of this work is to provide a unifying framework designed specially for programming high-performance computations on HNOCs. We seek to demonstrate that HMPI can provide a simple programming approach, portable applications, efficient performance, and predictable execution on HNOCs. Our results fall into the following categories:

- Extensions to MPI.
- Heterogeneous Data Partitioning Interface (HDPI).
- HMPI application programming.
- Programming environment to support HMPI programming.
- Experimentation demonstrating the effectiveness of HMPI in writing portable and efficient parallel applications.

The rest of the thesis addresses each of the above contributions. Chapter 2 provides a survey of related work. Chapter 3 presents the model of HMPI. The high-level architectural details of a research implementation of HMPI are also presented in this chapter. The classification of partitioning problems and the Heterogeneous Data Partitioning Interface (HDPI) are presented in Chapter 4. The methodology and features of the HMPI library are illustrated with some representative parallel HMPI applications in Chapter 5. Results of experiments with these applications on HNOCs investigate the merits of using HMPI. Conclusions and directions for future work follow in Chapters 6 and 7.

CHAPTER 2

Related Work

The section surveys related work from the literature, which includes mainly:

- High-level tools facilitating the implementation of parallel algorithms on distributed-memory architectures.
- Extensions to Message Passing Interface (MPI) and runtime systems that distribute data efficiently and automatically when there are changes in the application or the underlying environment.
- Implementations of MPI that adopt runtime adaptation schemes to find an efficient data distribution when workload and communication characteristics of a program change at runtime.
- Research dealing with a combined approach of compile-time analysis, runtime load distribution, and cooperation of operating system scheduler for improved utilization of resources on HNOCs.
- Data partitioning algorithms for mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) solving problems on HNOCs and performance models used for such data partitioning algorithms.
- Performance models of parallel architectures, execution-time estimation models for HNOCs, and models analysing the scalability of heterogeneous parallel algorithms.
- Static and dynamic mapping strategies used for matching and scheduling of application tasks to the machines.
- High performance computing on global networks.

2.1 High-level Parallel Programming Tools

The parallel programming tools developed to facilitate the implementation of parallel algorithms on distributed memory-architectures include (MPI) [SOJ+96], Parallel Virtual Machine (PVM) [GBD+94], High Performance Fortran (HPF) ([HPF94], [HPF97]), Dataparallel-C [HQ91], and mpC [AKL+99, LAK+00, Las02].

PVM and MPI are the main programming tools used for programming high-performance computations on homogeneous distributed-memory computer systems such as supercomputers and clusters of workstations. They are also normally used to write parallel programs for heterogeneous networks of computers (HNOCs). They are message passing packages providing, in fact, the assembler level of parallel programming for HNOCs.

The most important new concept introduced by MPI is the *communicator*. The communicator allows the programmer to safely separate messages that do not have to be logically mixed, even when the messages are transferred between processes of the same group. Logically a communicator may be seen as a separate communication layer associated with a group of processes. There may be several communicators associated with the same group, providing nonintersecting communication layers. It is this very feature that allows the programmer to use MPI for writing parallel libraries. In other words, the programmer can write an MPI subprogram that can be safely used by other programmers in their MPI programs without any knowledge of the details of its implementation. In contrast, PVM does not have the capacity to separate safely communication layers for message passing, and therefore it cannot be used for implementation of parallel libraries. The point is that the only unique attribute characterizing a PVM process is its ID assigned at runtime to each process of the PBM program. All other communication attributes, which could be used to separate messages, such as groups and tags, are user-defined. Therefore they do not have to be unique at runtime, especially if different modules of the program are written by different programmers.

Central to the design of PVM is the notion of a "virtual machine" – a set of heterogeneous hosts connected by a network that appears logically to the user as a single large parallel computer. The virtual machine in PVM also serves to encapsulate and organize resources for parallel programs. Further, this resource abstraction is carefully layered to allow varying degrees of control. The user might create an arbitrary collection of machines and then treat them as uniform computational nodes, regardless of their architectural differences. Although MPI does not have a concept of a virtual machine, MPI provides a higher level of abstraction on top of the computing resources in terms of message passing topology. In MPI a group of tasks can be arranged in a specific logical interconnection topology. A clear distinction must be made between the virtual process topology and the topology of the underlying, physical hardware. The virtual topology can be exploited by the system in the assignment of processes to physical processors.

However, these tools do not provide features to facilitate the development of adaptable parallel applications: that is, such applications that distribute computations and communications in accordance with input data and the peculiarities of the executing heterogeneous network. Even the topological facilities of the MPI have turned out to be insufficient to solve the problem. So, to ensure the efficient execution of the program on a particular network, the user must use facilities external to the program, such as boot schemes and application schemes [BDV94]. If the user is familiar with both the topology (that is, the structure and processor/link performances) of the target network and the topology (that is, the parallel structure) of the application, then, by means of use of such configuration files, he or she can map the processes, which constitute the program, onto processors of the network to provide the most efficient execution of the program. Some tools that support and facilitate such a static mapping have appeared, but if the application topology is defined at run time (that is, if it depends on the input data) this approach will not work.

The standard MPI specification does not provide tools that address some additional challenges posed by HNOCs, which are outlined below:

- **Heterogeneity of processors**. A good parallel application for HNOCs must distribute computations unevenly taking into account the speeds of the processors. The efficiency of the parallel application also depends on the accuracy of estimation of the speeds of the processors of the HNOCs, which is difficult because the processors may demonstrate different speeds for different applications due to differences in the set of instructions, the number of instruction execution units, the number of registers, the structure of memory hierarchy and so on.
- **Ad hoc communication network**. The common communication network is normally heterogeneous. The latency and bandwidth of communication links between different pairs of processors may differ significantly. This makes the problem of optimal distribution of computations and communications across the HNOCs much more difficult than across a dedicated cluster of workstations interconnected with a homogeneous high-performance communication network. Other issue is that the common communication network can use multiple network protocols for communication between different pairs of processors. A good parallel application should be able to use multiple network protocols between different pairs of processors within the same application for faster execution of communication operations.

- **Multi-user decentralized computer system**. Unlike dedicated clusters and supercomputers, HNOCs are not strongly centralized computer systems. A typical HNOC consists of relatively autonomous computers, where each one may be used and administered independently by its users. The first implication with the multi-user decentralized nature of HNOCs is the unstable performance characteristics of processors during the execution of a parallel program as the computers may be used for other computations and communications. The second implication is the much higher probability of resource failures in HNOCs compared to dedicated cluster of workstations, which makes fault tolerance a necessary feature for parallel applications running on HNOCs.

Thus, there are three important challenges (though these are not the only ones) posed by HNOCs, which are not addressed by the standard MPI specification.

Firstly, the standard MPI does not provide a means for employment of multiple network protocols between different pairs of processors for efficient communication in the same MPI application. A standard implementation of MPI does not address the challenge either. There have been majority of vendor implementations addressing this issue especially the use of shared memory and TCP/IP in MPICH [GLD+96], the support for multiple communication mediums (but not more than one device simultaneously) TCP, SMP, Myrinet, and InfiniBand in MPI/Pro [RS99, Dim01], and support of multiple communication devices simultaneously in WMPI [MS98b]. At the same time, there have been some research efforts to address this challenge implicitly, via advanced non-standard implementations of the standard MPI specification (Nexus [FGK+97], Madeleine [ABN00]).

Secondly, the standard MPI does not provide a means for the writing of fault-tolerant parallel applications for HNOCs. There are some research efforts made recently to address this challenge such as MPI-FT [LNL+00], MPI/FT [BNC+01], and the fault-tolerant MPI (FT-MPI) [FBD01]. MPI-FT proposes a fault tolerant and recovery scheme for MPI, consisting of a detection mechanism for detecting process failures and a recovery mechanism. The recovery function simulates all the communication of the processes with the dead one by re-sending to the replacement process all the messages destined for the dead one. In MPI-FT, each process keeps a buffer with its own message traffic, or a monitoring process, called the Observer, receives and stores all message traffic. MPI/FT is a high-performance MPI-1.2 implementation enhanced with low-overhead functionality to detect and recover from process failures. FT-MPI is also an MPI-1.2 specification implementation that provides process level fault tolerance at the MPI API level. FT-MPI survives the crash of **n-1** processes in an **n**-process job, and, if required, can respawn/restart them. It allows the application to continue using a communicator with the failed rank while explicitly excluding communication with the failed rank, or to shrink the communicator by excluding the failed rank, or to spawn a new process to take the place of the failed process. However, it is still the responsibility of the application to recover the data-structures and the data on the crashed processes.

Thirdly, the standard MPI does not provide features, which facilitate the writing of parallel programs that distribute computations and communications unevenly, taking into account the speeds of the processors, and the latencies and bandwidths of communication links.

High Performance Fortran (HPF) is a high-level parallel language that was originally designed for (homogeneous) supercomputers as the target architecture. HPF was standardized in 1994 as HPF 1.1 [HPF94]. It only provided regular mapping patterns and did not support uneven distribution of the array elements over processors, necessary for balanced mappings of heterogeneous algorithms. HPF 2.0 [HPF97] was standardized in 1997. HPF 2.0 provides extended mapping features that permit greater control over the mapping of data, including facilities for dynamic realignment and redistribution of arrays at runtime (REALIGN, REDISTRIBUTE, DYNAMIC directives), mapping of data among subsets of processors, mapping of pointers and components of derived types, and support for irregular distribution of data (GEN_BLOCK and INDIRECT distributions). The "generalized" block distribution, GEN_BLOCK, allows contiguous segments of an array, of possibly unequal sizes, to be mapped onto processors. The INDIRECT distribution allows a many-to-one mapping of elements of a dimension of a data array to a dimension of a target processor arrangement. Thus, HPF-2 provides some basic support for programming heterogeneous algorithms. At the same time, HPF-2 provides no language constructs allowing the programmer to better control mapping of the heterogeneous algorithms to HNOCs. The HPF programmer has to rely on some default mapping provided by the HPF compiler.

Dataparallel-C is an extension to ANSI C that allows programmers to write efficient code for parallel systems. It brings the data parallel programming model to C. It presents the model of computation characterized by a global name space, synchronous execution, and virtual processors as the unit of parallelism. Virtual processors are allocated in groups of like type. Each virtual processor in the group has an identical memory layout. The Dataparallel C programmer specifies a virtual processor's memory layout using syntax similar to the C struct using a new keyword `domain`. Dataparallel C allows additional information to be provided by the programmer in order to aid the compiler in the mapping of virtual processors to physical processors. The array dimension of the domain array establishes a *virtual topology*. A one-dimensional domain array is considered to be a ring of virtual processors; a two-dimensional domain array is considered to be a torus (doughnut topology). Dynamic load balancing is accomplished through periodic exchange of load information during calls to the routing library at runtime.

The mpC language is an ANSI C superset designed specially for programming high-performance computations on HNOCs. The main idea underlying mpC language, is to provide language constructs that allow the user to define in detail an abstract heterogeneous parallel machine that is most appropriate to his/her parallel algorithm. The mpC language allows the programmer to define at runtime all the main features of parallel algorithm, which have an impact on the execution performance of the application on heterogeneous platforms, including:

- The total number of processes executing the algorithm,
- The total volume of computations to be performed by each of the processes in the group during the execution of the algorithm,
- The total volume of data to be transferred between each pair of processes in the group during the execution of the algorithm, and
- The order of execution of the computations and communications by the involved parallel processes in the group, that is, define exactly how the processes interact during the execution of the algorithm.

Such an abstraction of parallel algorithm is called a *network type*. The mpC programming system uses the information extracted from the definition of network type together with information about actual performances of processors and communication links of the executing network to map the processes of the parallel program to this network in such a way that better execution time is achieved. The most important features of mpC are:

- once developed, an mpC application will run efficiently on HNOCs without any changes to its source code.
- it allows one to write applications adapting not only to nominal performances of processors but also to redistribute computations and communications dependent on dynamic changes of workload of separate computers of the executing network.

The mpC language provides all the facilities lacking in the other high-level tools. It expresses both data and control parallelism. Like HPF, it includes a vector subset, named the C[] language [GL94] to provide data parallelism. It provides implicit communication through message passing, that is, the programmer does not have to program the communication. It also supplies mechanisms, in the form of network types, to the user to guide the mapping process essential for exploiting his or her knowledge of the application.

2.1.1 Summary

The various programming environments for HNOCs surveyed, excluding mpC, lack either the facilities to describe the virtual parallel system, or such facilities are too poor to specify an efficient distribution of computations and communications over the target network. The problems with mpC are mainly to do with the learning curve associated with a high-level parallel language. The other issue is the limitation on its adaptability to unanticipated machines, algorithmic models and data structures. In many cases these require new semantics to be added to the language for efficient implementations to be automatically generated.

2.2 MPI Extensions/Implementations for HNOCs

Dyn-MPI [WLN+03] extends MPI by providing specialized facilities for memory allocation, communication, and node participation. The key component of Dyn-MPI is its run-time system, which efficiently and automatically redistributes data on the fly when there are changes in the application or the underlying environment. Dyn-MPI also provides a facility for removing nodes from the computation when their participation degrades performance. However converting an MPI program to a Dyn-MPI program can require major modifications for it to run efficiently on HNOCs. These include allocation functions for all potentially re-distributable arrays and determination of Deferred Regular Section Descriptors (DRSDs) for each array. Also the computational model of Dyn-MPI is essentially Single Program Multiple Data (SPMD), in which each node executes the same program text but will take different execution paths through this text depending on the input data.

AMPI [BKS+01, LBK02] is an implementation of a significant subset of MPI 1.1 Standard over CHARM++ [KK93]. AMPI utilizes the dynamic load-balancing capabilities of CHARM++ by associating a “user-level” thread with each CHARM++ migratable object. User’s code runs inside this thread, so that it can issue blocking receive calls similar to MPI, and still present the underlying scheduler an opportunity to schedule other computations on the same processor. The runtime system keeps track of computation loads of each thread as well as communication graph between AMPI threads, and can migrate these threads in order to balance the overall load while simultaneously minimizing communication overhead. However converting an MPI program to an AMPI program can require major modifications for it to run efficiently on HNOCs. These include privatization of global variables, registering chunk data and providing a packing subroutine to the AMPI runtime system to pack the thread’s data, and the migration decision has to be made by the user through a call to migration subroutine even though the actual migration of the chunk is done by the system’s load balancing strategy.

Tern [KS01] is an implementation of a subset of the MPI standard for message passing parallel programming, and augments the MPI standard to provide for multithreaded execution. It provides the ability to transparently migrate threads between the nodes executing the parallel application thus achieving parallel program performance improvement through load balance and improved fault tolerance. Tern makes use of a novel lazy heap migration protocol, which can greatly reduce the amount of time necessary to migrate a thread and its corresponding data regions. However in a threaded environment, static and global variables will be shared among all threads executing in the same process and so need to be made thread-specific. Tern provides a compiler directive prefix that is required to be added to each global variable and static-local variable declaration. This allocates thread-specific instances of all variables utilizing the concept of thread-local storage. Tern exposes the policy mechanisms used to guide migration decisions to the user, allowing for customizable thread migration policies. Tern provides two mechanisms for safely migrating a user thread. The first mechanism allows the user to insert migration calls in the user thread. However this mechanism requires that the user knows exactly where to insert the migration calls. Alternatively, Tern runtime system migrates the thread depending on the policy mechanisms devised by the user.

CRAUL [RD01] is a runtime system that combines compile-time analysis, runtime load balancing and locality considerations, and cooperative scheduling support from the operating system for improved performance of parallel programs on HNOCs. The CRAUL compiler is a Stanford University Intermediate Format (SUIF) compiler [AAL+95] with two additional passes. The first pass works before the parallel code generation and inserts code with access information about each parallel region's access patterns. The second pass works on parallelized programs and modifies the loop structure so that a task queue is used. The runtime system uses this loop and accesses information to partition the available work based on locality of data access as well as resource availability. The operating system responds to application-specific information on scheduling needs while respecting fairness. The operating system also provides feedback to the application about the scheduling status of the cooperating processes, allowing the runtime to make resource management decisions based on this information.

However, the CRAUL compiler lacks the more complex translation mechanisms essential to extract parallelism from less easily analyzable loops. In such cases, the user needs to insert the required data structures manually into an already parallelized program. The problem of portability and reusability of the parallel code generated is not addressed. Also CRAUL does not provide features required to capture programmer's knowledge of an application to the level necessary to automatically provide an efficient implementation on a heterogeneous system. CRAUL shields the user from data distribution details but does not supply mechanisms for the user to guide the mapping process essential for exploiting his or her knowledge of the application.

2.2.1 Summary

The various extensions to MPI and its implementations surveyed shield the user from data distribution details. These research efforts provide compiler and runtime systems that perform the tedious and error-prone chore of load balancing. However the tasks of mapping of the parallel processes to the executing heterogeneous network and scheduling are not addressed. These tools also do not provide mechanisms to the programmer to guide the load balancing, mapping and scheduling processes that can exploit his or her knowledge of the application. Instead they tend to automatically discover the algorithmic properties from the code, which is non-trivial in many aspects. Ideally a tool must supply mechanisms to the programmer so that he or she can provide information to it that can assist in finding the most efficient implementation on a HNOCs. Combining the system's detailed analysis with the programmer's high-level knowledge of the application is essential in finding more efficient mappings than either one alone is capable of achieving.

2.3 Data Partitioning

The core of scientific, engineering or business applications is the processing of some mathematical objects that are used in modeling corresponding real-life problems. In particular, partitioning of such mathematical objects is a core of any data parallel algorithm. Our analysis of various scientific, engineering and business domains resulted in the following short list of mathematical objects commonly used in parallel and distributed algorithms: **sets** (ordered and non-ordered), **dense matrices** (and multidimensional arrangements) and **sparse matrices**, **graphs**, and **trees** (a tree is a graph in which any two vertices are connected by *exactly one* path).

Sets

A **set** is a well-defined collection of objects considered as a whole. The objects of a set are called elements or members. We consider the elements of the set to represent independent chunks of computations, each of equal size (i.e., each requiring the same amount of work), which can be computed without reference to each other i.e., without communication.

There are two main criteria used for partitioning a set:

1) The number of elements in each partition should be proportional to the speed of the processor owning that partition.
2) The sum of weights of the elements in each partition should be proportional to the speed of the processor owning that partition.

Additional restrictions that may be imposed on partitioning of an **ordered set** are:

- The elements in the set are well ordered and should be distributed into disjoint contiguous chunks of elements.

The most general problem of partitioning a **set** can be formulated as follows:

- Given: (1) A set of n elements with weights w_i (i=0,…,n-1), and (2) A well-ordered set of p processors whose speeds are functions of the size of the problem **x** (We define the size of the problem to be the amount of data stored and processed by the sequential algorithm), $s_i=f_i(x)$, with an upper bound b_i on the number of elements stored by each processor (i=0,…,p-1),
- Partition the set into p disjoint partitions such that: (1) The sum of weights in each partition is proportional to the speed of the processor owning that partition, and (2) The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it.

The most general partitioning problems for a **set** and an **ordered set** are very difficult and open for research.

One example of a special partitioning problem for a set is:

- Given: (1) A set of n elements, (2) A well-ordered set of p processors whose speeds are represented by single constant numbers, $s_0,s_1,…,s_{p-1}$, and (3) There are no limits on the maximal number of elements assigned to a processor,
- Partition the set into p disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that partition.

The algorithm used to perform the partitioning is quite straightforward, of complexity O($\mathbf{p}^2$) [BBP+01]. The algorithm uses a naive implementation. The complexity can be reduced down to O($\mathbf{p} \times \log_2 \mathbf{p}$) using ad hoc data structures [BBP+01].

Dense Matrices

Matrix partitioning algorithms are usually designed during the implementation of heterogeneous parallel algorithms for linear algebra ([CQ93], [KL01], [BBR+01], [BBP+01]). These are modifications of traditional homogeneous algorithms with mappings for HNOCs. The mappings take into account all peculiarities of the corresponding parallel algorithms and are based on very careful performance analysis. The typical partitioning of a matrix uses block-cyclic distribution of matrices on either a one-dimensional or on a two-dimensional grid of processors. Blocked versions of the parallel algorithms for matrix multiplication and linear system solvers are used in ScaLAPACK (Scalable Linear Algebra Package) [CDD+96] to squeeze the most out of state-of-the-art processors with pipelined arithmetic units and multilevel memory hierarchy. The block cyclic distribution has been also incorporated in the HPF language [HPF97].

Sparse Matrices

A **sparse matrix** is a matrix populated primarily with zeros. More precisely, a matrix is considered sparse if a computation involving it can utilize the number and location of its nonzero elements to reduce the run time over the same computation on a dense matrix of the same size. It is customary to store an $n \times n$ matrix in an $n \times n$ array. However, if the matrix is sparse, storage is wasted because a majority of the elements of the matrix are zero and need not be stored explicitly. For sparse matrices, it is common practice to store only the nonzero entities and to keep track of their locations in the matrix. A variety of storage schemes are used to store and

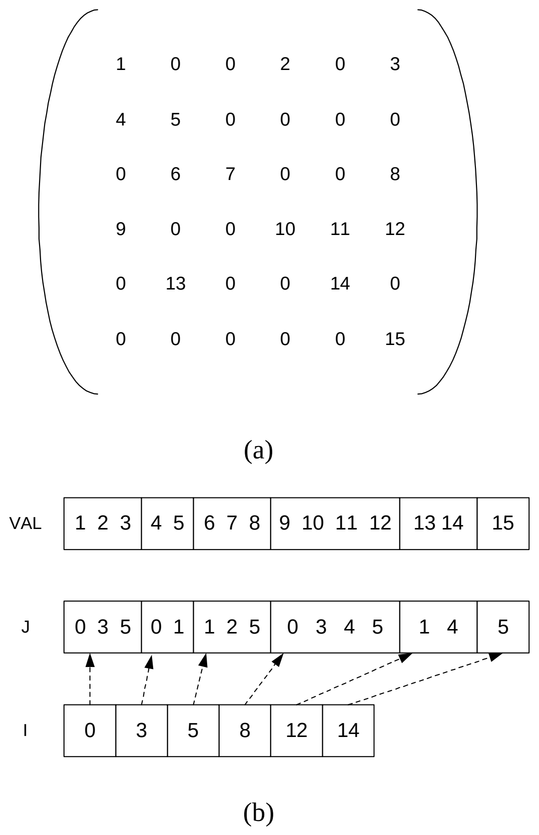


Figure 2.1: A 6×6 sparse matrix and its representation in Compressed Storage Row (CSR) format.

manipulate sparse matrices [KGG+94]. There is no single best data structure for storing sparse matrices. Different data structures are suitable for different operations.

There are two methods to partition a sparse matrix:

- The application of set partitioning algorithms to the data storage scheme used for storing a sparse matrix. For example, assuming that a sparse matrix is stored in CSR (Compressed Sparse Row) format. In CSR format illustrated in Figure 2.1, there are three arrays to store an $n \times n$ sparse matrix with q nonzero entries: 1). A $q \times 1$ array VAL contains the nonzero elements. These are stored in the order of their rows. 2) A $q \times 1$ array J that stores the column numbers of each nonzero element. 3) An $n \times 1$ array I, the i-th entry of which points to the first entry of the i-th row in VAL and J. The sparse matrix is then partitioned such that the number of elements in the array *VAL* (the array is assumed to be an ordered set) is proportional to the speed of the processor.

- Transformation of a sparse matrix into a graph and application of graph partitioning algorithms to the graph.

Balay *et al*. [BGM+97] propose PETSc, which is a suite of data structures and routines for scalable (parallel) solution of scientific applications modeled by partial differential equations. PETSc supports variety of sparse storage formats because no single sparse storage format is appropriate for all problems.

Birov *et al*. [BPB+99] present a parallel mathematical library suite for sparse matrices. The Parallel Mathematical Libraries Project (PMLP) constitutes a concerted effort to create a supportable, comprehensive "Sparse Object-oriented Mathematical Library Suite." PMLP includes operations on various matrix types such as general, banded, symmetric, banded symmetric, skew symmetric, hermitian, skew hermitian, and lower and upper triangular matrices. Furthermore, PMLP provides functionality independent of the internal data representation of irregular sparse objects and different storage matrix formats (e.g. coordinate, compressed sparse column, compressed sparse row, sparse diagonal, dense, Ellpack/Itpack, and skyline) are included.

Graphs

The standard graph partitioning approach has been to divide the vertices of the graph into approximately equal-weight partitions (balance computations) and minimize the number of cut edges between partitions (minimize total runtime communication). Formally, the *k*-**way** graph partitioning problem is defined as follows: Given a graph G=(V,E) with $|V| = n$, partition V into k subsets, $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \varnothing$ for $i \neq j$, $|V_i| = n/k$, and $\bigcup_i V_i = V$, and the number of

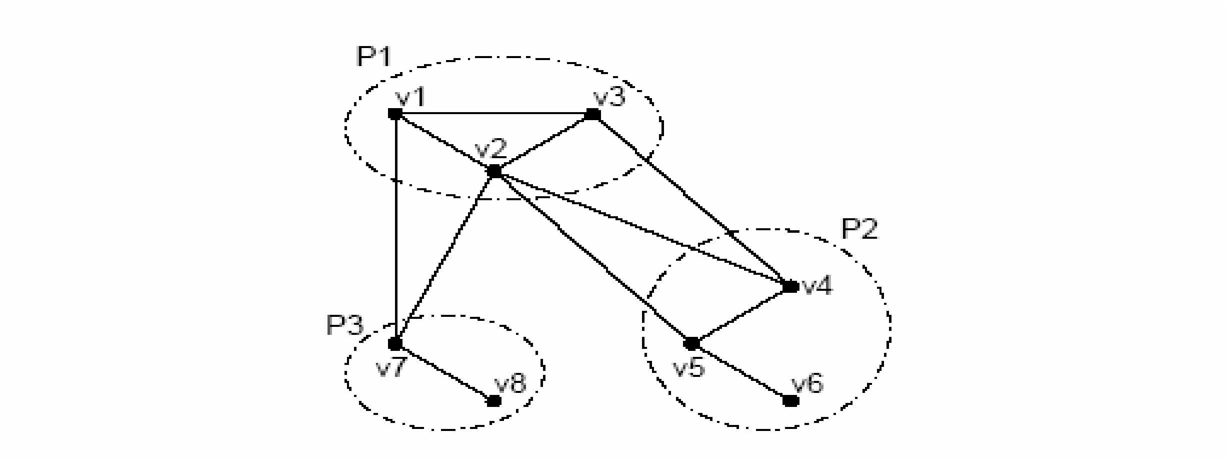


Figure 2.2: Edge cuts versus communication volume.

edges of *E* whose incident vertices belong to different subsets is minimized. The *k*-way graph partitioning problem can be naturally extended to graphs that have weights associated with the vertices and the edges of the graph. In this case, the goal is to partition the vertices into *k* disjoint subsets such that the sum of the vertex-weights in each subset is the same, and the sum of the edge-weights whose incident vertices belong to different subsets is minimized. A *k*-way graph partition of *V* is commonly represented by a partition vector *P* of length *n*, such that for every vertex $v \in V$, $P[v]$ is an integer between 1 and *k*, indicating the partition to which vertex *v* belongs. Given a partition *P*, the number of edges whose incident vertices belong to different subsets is called the ***edge-cut*** of the partition.

Unfortunately, the standard graph partitioning approach has several significant shortcomings. Firstly, the edge cut metric that it tries to minimize is, at best, an imperfect model of communication in a parallel computation. Edge cuts are *not* proportional to the total communication volume as illustrated in Figure 2.2. The ovals correspond to different processors among which the vertices of the graph are partitioned. Assume that each edge has a weight of two corresponding to one unit of data being communicated in each direction, so the weight of the cut edges is ten. However, observe that the data from node v2 on processor P1 need only be communicated once to processor P2; similarly with nodes v4 and v7. Thus, the actual communication volume is only seven. In general, the edge cut metric does not recognize that two or more edges may represent the same information flow, so it overcounts the true volume of communication. The model also suffers from a lack of expressibility that limits the applications it can address.

Secondly, the time to send a message on a parallel computer is a function of the latency (or start-up time) as well as the size of the message. As has been observed by a number of researchers, graph partitioning approaches try to (approximately) minimize the total volume but not the total number of messages. Depending on the machine architecture and problem size, message latencies can be more important than message volume.

Third, the performance of a parallel application is generally limited by the slowest processor. Even if the computational work is well balanced, the communication effort might not be. Rather than minimizing the total communication volume or even the total number of messages, we may instead wish to minimize the maximum volume and/or number of messages handled by any single processor. As several researchers have noted, the standard edge cuts measure does not encapsulate this type of objective.

The standard model using an undirected graph can only encode symmetric data dependencies and symmetric partitions. These limitations are a particular problem for iterative solvers on unsymmetrical and non-square matrices. Hendrickson and Kolda [HK00] propose a bipartite graph model (A bipartite graph is a special graph where the set of vertices can be divided into two disjoint sets with two vertices of the same set never sharing an edge) for describing matrix-vector multiplication that addresses some of these shortcomings. The bipartite model can also be applied to other problems involving unsymmetrical dependencies and multiple phases.

Edge cuts are not equal to communication volume, as illustrated in Figure 2.2. The true communication volume is not a function of the number of edges being cut, but rather the sum of the number of processors to which each vertex has connections. More formally, Given a graph $G=(V,E)$ with $|V| = n$, and a partition vector P of length n such that $P[v]$ stores the number of the partition that vertex v belongs to. Let $V_b \subset V$ be the subset of interface vertices. That is, each vertex $v \in V_b$ is connected to at least one vertex that belongs to a different partition. For each vertex $v \in V_b$ let $Nadj[v]$ be the number of domains other than $P[v]$ that the vertices adjacent to v belong to. The total communication volume is defined as $\sum_{v \in V_b} Nadj[v]$. This is the total communication volume incurred by the partitioning because each interface vertex v needs to be sent to all of its $Nadj[v]$ partitions. This is also called the *boundary cut* of the partition. In particular, if w_v is the amount of data that needs to be sent for vertex v, then the boundary cut is $\sum_{v \in V_b} w_v \times Nadj[v]$. Minimizing boundary cuts is a non-traditional graph partitioning problem, but it can be addressed using the same algorithmic tools that have been developed for other partitioning variants. A more elegant expression of this metric is in the hypergraph model proposed by [CA96, PCA+96]. By partitioning the hypergraph so that hyperedges are split among as few processors as possible, the model correctly minimizes communication volume.

The constraint partitioning model proposed by Karypis and Kumar [KK98a] can be used for multi-phase calculations. In the multi-constraint model, each vertex is assigned a vector of m weights that represent the work associated with that vertex in each of the m computational phases. The goal is to partition the vertices of that graph in such a way that communication is minimized and each of the *m* weights is balanced. In this way, each phase of the computation will be load balanced. The goal is to compute a *k*-way partitioning such that each one of the *m* weights is individually balanced within a specified tolerance. As an example, consider the multiphysics simulation (Multiphysics simulation is based on a single computational framework for the modeling of multiple interacting physical phenomena) in which the amount of computation as well as the memory requirements is not uniform across the mesh. Existing single-constraint graph partitioning algorithms allow us to easily partition the mesh among the processors such that either the amount of computations is balanced or the amount of memory required by each partition is balanced; however, they do not allow to compute a partitioning that simultaneously balances both of these quantities.

A related model is the multi-objective approach of Schloegel *et al*. [SKK99a]. This model attempts to address the common situation in which a partition should simultaneously minimize several cost functions. To achieve this, each edge is given a vector of *m* weights, each of which reflects one of the *m* different cost functions. The goal of the partitioning is to balance the vertex weights in such a way that each of the cost functions is kept small. They discuss two ways of disambiguating the definition of a good multi-objective solution, which are (i) to prioritize the objectives, and (ii) to combine the objectives into a single objective. In the priority based formulation, the user is allowed to assign a priority ranging from one to *m* to each of the *m* objectives. The multi-objective partitioning problem becomes that of computing a *k*-way partitioning such that it simultaneously optimizes all the *m* objectives, giving preferences to the objectives with higher priorities. In the combination-based formulation, multiple objectives are combined into a single objective and then the single objective optimization technique is used.

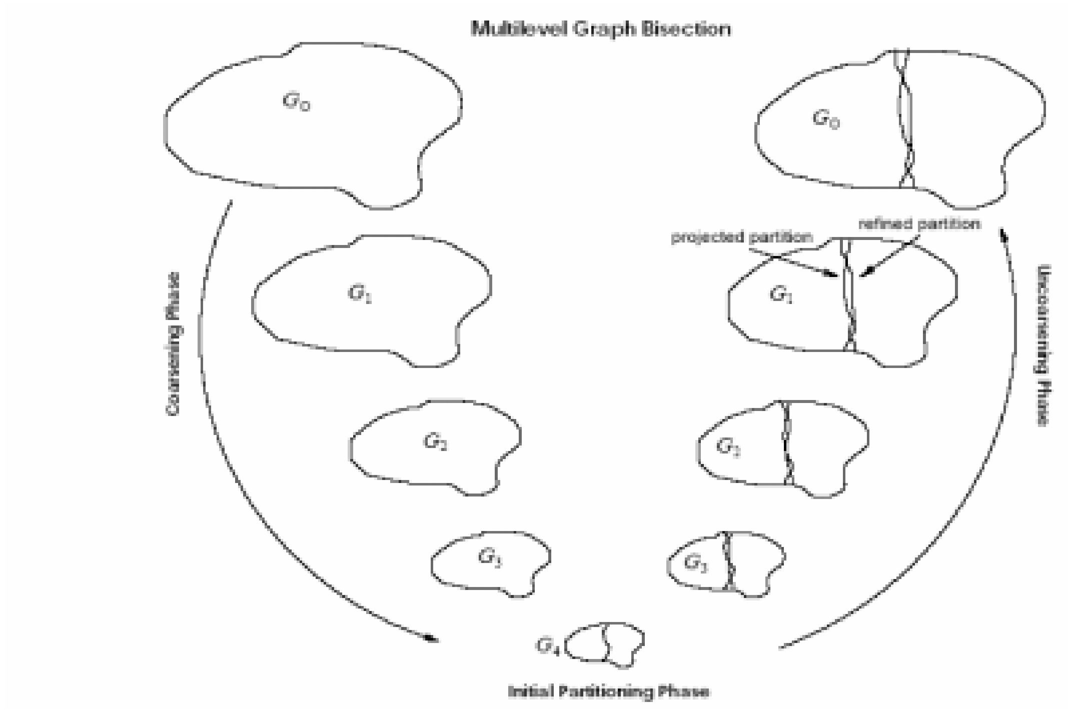


Figure 2.3: The various phases of the multilevel graph bisection. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening phase, the bisection is successively refined as it is projected to the larger graphs. During the uncoarsening phase the light lines indicate projected partitions, and dark lines indicate partitions that were produced after refinement.

Typically, this is done by taking the sum of the elements of the objective vector weighted by a *preference* vector, *p*.

In the skewed model developed by Pellegrini [Pel94] and Hendrickson *et al.* [HLD97], each vertex is allowed a set of *k* preference values expressing its relative desire to be in each of the *k* sets. When partitioning a graph, let each vertex *i* have a *desire* to be in set *k* denoted by $d_k(i)$. The goal then is to minimize the cut edges and maximize the satisfied desires. Let $s(i)$ be the set to which vertex *i* is assigned. We want to find a mapping *s*, which minimizes the following objective function.

$$\text{Minimize} \quad \sum_{e_{ij}} \begin{cases} w_e(e_{ij}) & \text{if } s(i) \neq s(j) \\ 0 & \text{otherwise} \end{cases} - \sum d_{s(i)}(i) \;\forall\; i = 0, 1, \ldots, \boldsymbol{n}$$

where e_{ij} denotes an edge between vertices i and j and $w_v(i)$ and $w_e(e_{ij})$ representing weights of vertices and edges respectively.

The different graph partitioning models described above are only viable if efficient and effective algorithms can be developed to partition them. Spectral partitioning algorithms are known to produce good partitions for a wide class of problems, and they are used extensively [PSL90, HL93b, BS89]. Geometric partitioning algorithms use the geometric information of the graph to find a good partition [Rag93, MTV91]. Multilevel algorithms [BJ93, HL93a] have been a universal approach to solving the graph partitioning problem on homogeneous networks of computers. There are three different stages to multilevel graph partitioning algorithms as shown for multilevel graph bisection in Figure 2.3. First, a sequence of smaller and smaller graphs is created from the original graph. Second, the smallest graph is partitioned. And third, the partition is propagated back through the sequence of graphs, with an occasional local refinement. Some of the options applicable to the three phases in the multilevel partitioning algorithms are outlined below:

(1). *Coarsening Phase*

- Procedure to determine the maximal matching. We define a matching of a graph G as a subset E_m of the edges E with the property that no two edges in E_m share an endpoint. A maximum matching of graph G is a matching of G with the greatest number of edges while a maximal matching is a matching which is not contained in any larger matching. While any maximum matching is certainly maximal, the reverse

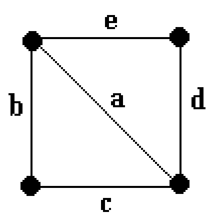


Figure 2.4: A sample graph.

is not generally true. In the graph shown in Figure 2.4, {a}, {b,d}, and {c,e} are all maximal matchings, but only the last two are maximum matchings.

- Random Edge Matching (RM) [BJ93, HL93a],
- Heavy Edge Matching (HEM) [KK95],
- Light Edge Matching (LEM) [KK95],
- Heavy Clique Matching (HCM) [KK95].

- Node reduction between successive coarsening levels,
- Maximum number of vertices in the coarsest graph.

(2). *Partitioning Phase*

- Spectral Bisection (SB) [BS89, HL93a],
- Kernighan-Lin Algorithm (KL)/Fiduccia-Mattheyses [KL70, FM82],
- Graph Growing Algorithm (GGP) [GL81, GS94, CL94],
- Greedy Graph Growing Algorithm (GGGP) [KK95].

(3). *Uncoarsening Phase*

- Kernighan-Lin Refinement [KL70, HL93a, KK95],
- Boundary Kernighan-Lin Refinement [KL70, HL93a, KK95].

METIS [KK95] is a set of programs for partitioning graphs, partitioning finite element meshes, and for producing fill reducing orderings for sparse matrices. The algorithms implemented in METIS are based on the multilevel graph partitioning schemes. ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel Adaptive Mesh Refinement (AMR) computations and large scale numerical simulations. The algorithms implemented in ParMETIS are based on the parallel multilevel *k*-way graph-partitioning algorithms described in [KK97], the adaptive repartitioning algorithm described in [SKK00], and the parallel multi-constrained algorithms described in [SKK99b]. hMETIS is a set of programs for partitioning hypergraphs such as those corresponding to VLSI circuits. The algorithms implemented by hMETIS are based on the multilevel hypergraph partitioning scheme described in [KAK+97] and [KK98b].

Chaco [HL94] contains a variety of graph partitioning algorithms including spectral bisection, quadrisection and octasection, the inertial method, the Kernighan-Lin/Fiduccia-Mattheyses algorithm and multilevel partitioners. Advanced techniques that are new to version 2.0 include terminal propagation (a method for improving data locality adapted from the circuit community), the ability to map partitions intelligently to hypercube and mesh architectures, and easy access to the Fiedler vector to assist the development of new applications of spectral graph algorithms.

JOSTLE [WC00] is a software package designed to partition unstructured meshes (for example, finite element or finite volume meshes) for use on distributed memory parallel computers. It can also be used to repartition and load-balance existing partitions (such as those deriving from adaptive refined meshes). It achieves this by modeling the mesh as an undirected graph and then using state-of-the-art graph partitioning techniques. JOSTLE takes into account heterogeneous CPU performance using integral load-balancing capabilities. For example, given a graph of say 75 vertices and two processors, with processor 1 twice as fast as processor 2, the user may impose a penalty weight (based on the relative speeds and the total vertex weight; in this case 25) on processor 2 to simulate its slower performance. The load-balancer within JOSTLE then balances the total graph weight plus any penalty weights (in this example 75+25=100) and gives an equal share (50) to each processor. Because processor 2 has a penalty weight of 25, its share of the vertices is 25 as compared with the 50 of processor 1 and so the partition is balanced to reflect the relative performance of the processors.

Walshaw and Cross [WC01] modify the multilevel algorithms in order to minimize a cost function based on a model of the communications network supplied by the user at runtime. They deal with networks in which the communications cost (both latency and bandwidth) is not uniform across the inter-processor network. Let $G=(V,E)$ be an undirected graph of vertices V, with edges E which represent the data dependencies in the mesh. To model the true communication cost and build the cost function, a weight is assigned to the link between every pair of processors giving a network N represented by a weighted graph $N(P, L)$, where P is the set of p processors and L the set of interprocessor edges which is complete (i.e., there is an edge for every pair of processors) and weighted. The contribution to the cost function from every cut edge (v,w) with $v\in S_p$ and $w\in S_q$ (S_p and S_q are the subdomains assigned to processors p and q respectively)is defined to be $|(v,w)|\cdot|(p,q)|$, the weight of the cut edge multiplied by the weight of the link over which it passes. Thus given a partition $\pi : V \to P$, the cost function is given by

$$\Gamma = \sum_{(v,w)\in E_c} |(v,w)|\cdot|(\pi(v),\pi(w))|$$

where E_c is the set of cut edges. The mapping problem can then be defined as: given a graph $G=(V,E)$ and a processor network $N(P, L)$, find $\pi : V \to P$, a mapping of vertices to processors, such that $\left|S_p\right| \leq \bar{S}$ where $\bar{S} := \left\lceil |V| / P \right\rceil$ for all subdomains S_p and such that Γ is minimized. They do not address the issue of heterogeneous CPU performance. This is taken care of by load balancer within JOSTLE. The mapping algorithm is then compared with the two stage approach of partitioning of the graph such that edge cut is minimized followed by mapping of subdomains to processors (also known as processor assignment).

Kumar *et al*. [KDB02] employ a multilevel heterogeneous partitioner developed for distributed heterogeneous systems that differs from existing partitioners in that it takes into account both the system and work load graphs. In their model, the heterogeneous system consists of processors with varying processing power and an underlying non-uniform communication network. The partitioning algorithm employed, called *Minimax*, generates and maps partitions onto a heterogeneous system with the objective of minimizing the maximum execution time of the distributed parallel application.

Trees

In graph theory, a **tree** is a graph in which any two vertices are connected by exactly one path. The most general problem of partitioning a tree can be formulated as follows: Given a tree T consisting of n vertices $\{0,1,\cdots,n-1\}$ with weights v_i $(i = 0,1,\cdots,n-1)$ and m edges $\{0,1,\cdots,m-1\}$ with weights e_j $(j = 0,1,\cdots,m-1)$ and given a linear array of p processors whose speeds are functions of the size of the problem x, $s_0 = f_0(x), s_1 = f_1(x), ..., s_{p-1} = f_{p-1}(x)$ and there is an upper bound b_k $(k = 0,1,\cdots,p-1)$ on the number of vertices that each processor can hold, partition the tree into p disjoint subtrees such that

- The sum of weights of the vertices in each partition (subtree) is proportional to the speed of the processor owning that partition.

- The number of vertices assigned to each processor does not exceed the upper bound on the number of vertices stored by it.
- The edgecut is minimal.

This is an open problem for research. The partitioning operations on graphs can be used to partition a tree into disjoint partitions when there is no restriction that all the disjoint partitions have to be subtrees.

Perl and Schach [PS81] present efficient algorithms for a max-min k-partitioning problem, which can be formulated as follows: Given a tree T with n vertices and m edges and a nonnegative weight associated with each vertex. Let $k < n$ be a positive integer. The problem is to delete k edges in the tree so as to maximize the weight of the lightest of the resulting connected subtrees.

Becker *et al*. [BPS82] present efficient algorithms for min-max k-partitioning problem, which can be formulated as follows: Given a tree T with n vertices and m edges and a nonnegative weight associated with each vertex. Let $k < n$ be a positive integer. The problem is to delete k edges in the tree so as to minimize the weight of the heaviest of the resulting connected subtrees.

Frederickson [Fre91] present linear time algorithms for partitioning a tree with weights on the nodes by removing k edges so as to either minimize the maximum weight component or maximize the minimum-weight component.

The data partitioning approaches described above use different performance models of HNOCs to distribute computations amongst the processors involved in their execution. All the models use a single positive number to represent the speed of a processor, and computations are distributed amongst the processors such that their volume is proportional to this speed of the processor. However these models are efficient only if the relative speeds of the processors involved in the execution of the application are a constant function of the size of the problem and can be approximated by a single number. This is true mainly for homogeneous distributed memory systems where:

- The processors have almost the same size at each level of their memory hierarchies, and
- Each computational task assigned to a processor fits in its main memory.

But these models become inefficient in the following cases:

- The processors have significantly different memory structure with different sizes of memory at each level of memory hierarchy. Therefore, beginning from some problem size, the same task will still fit into the main memory of some processors and stop fitting into the main memory of others, causing the paging and visible degradation of the speed of these processors. This means that their relative speed will start significantly changing in favor of non-paging processors as soon as the problem size exceeds the critical value.
- Even if the processors of different architectures have almost the same size at each level of the memory hierarchy, they may employ different paging algorithms resulting in different levels of speed degradation for the task of the same size, which again means the change of their relative speed as the problem size exceeds the threshold causing the paging.

Thus considering the effects of processor heterogeneity, memory heterogeneity, and the effects of paging significantly complicates the design of algorithms distributing computations in proportion with the relative speed of heterogeneous processors. One approach to this problem is to just avoid the paging as it is normally done in the case of parallel computing on homogeneous multi-processors. However avoiding paging in local and global HNOCs may not make sense because in such networks it is likely to have one processor running in the presence of paging faster than other processors without paging. It is even more difficult to avoid paging in the case of distributed computing on global networks. There may not be a server available to solve the task of the size you need without paging.

Therefore, to achieve acceptable accuracy of distribution of computations across heterogeneous processors in the possible presence of paging, a more realistic performance model of a set of heterogeneous processors is needed. This model must integrate the essential features underlying applications run on HNOCs, mainly, the speeds of the processors, the latency and the bandwidth of the communication links between the processors, the memory heterogeneity in terms of the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy, and the effects of paging.

2.3.1 Summary

From the survey on data partitioning, it can be concluded that no classification of partitioning problems currently exists. Only matrix and graph partitioning problems have been widely studied. It is to be noted also that the algorithms solving these problems use performance models that do not take into account all the essential features underlying applications run on HNOCs, mainly, the speeds of the processors, the latency and the bandwidth of the communication links between the processors, the memory heterogeneity in terms of the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy, and the effects of paging.

2.4 Performance Models of Parallel Computers

A good deal of theoretical research has focused on models of parallel computers. The most widely used parallel models are the parallel random access machine (PRAM) [FW78], the bulk-synchronous parallel model (BSP) [Val90], and the LogP model [CKP+93]. All the models assume a parallel computer to be a homogeneous multiprocessor.

The PRAM is the most simplistic parallel computational model. The PRAM model consists of *p* sequential processors sharing a global memory. During each time step or cycle, each processor executes a RAM instruction or accesses global memory. After each cycle, all processors implicitly synchronize to execute the next instruction. The PRAM model assumes that synchronization and communication is essentially cost free. However, these overheads can significantly affect algorithm performance. By ignoring costs associated with exploiting parallelism, the PRAM is a simple abstraction, which allows the designer to expose the maximum possible computational parallelism in a given task. Many modifications to the PRAM have been proposed that attempt to bring it closer to practical parallel computers.

The BSP model is a bridging model that consists of *p* parallel/memory modules, a communication network, and a mechanism for efficient barrier synchronization of all the processors. It is referred to as a bridging model because it lies between hardware and programming models. It is efficient both in implementing high-level language features and algorithms, as well as in being implemented in hardware. A computation consists of a sequence of supersteps. During a superstep, each processor performs synchronously some combination of local computation, message transmissions, and message arrivals. Three parameters characterize the performance of a BSP computer. *p* represents the number of processors, *L* measures the minimal number of time steps between successive synchronization operations, and *g* represents the minimum time interval between consecutive message transmissions on a per-processor basis. The total execution time for the program is the sum of all superstep times.

An approach related to BSP is the LogP model. LogP models the performance of point-to-point messages with three parameters: o (computation overhead of handling a message), g (time interval between consecutive message transmissions at a processor), and L (latency for transmitting a single message). The LogP model has been successfully used for developing fast and portable parallel algorithms for (homogeneous) supercomputers. LogGP [AIS+95] is an extension of LogP used to study the impact of long messages on algorithm design. LoGPC [MF98] models the network contention effects. PLogP model [KBV00] is an extension of the LogP model. PLogP model is defined in terms of end-to-end latency L, sender and receiver overheads, $o_s(m)$ and $o_r(m)$ respectively, gap per message $g(m)$, and number of nodes involved in communication P. In this model sender and receiver overheads and gap per message depend on the message size. Notion of latency and gap in PLogP model slightly differs from that of the LogP/LogGP model. Latency in PLogP model includes all contributing factors, such as copying data to and from network interfaces, in addition to the message transfer time. Gap parameter in PLogP model is defined as the minimum time interval between consecutive message transmissions or receptions, implying that at all times $g(m) >= o_s(m)$ and $g(m) <= o_r(m)$. Nonetheless, the LogP model or its extensions are inappropriate to model HNOCs, mainly due to their non-deterministic nature and irregularity.

There are a few computational approaches to support heterogeneous parallel computation, namely, Heterogeneous Coarse-grained Multicomputer (HCGM) [Mor98], Heterogeneous Bulk Synchronous Parallel (HBSP) [WP00], and k-Heterogeneous Bulk Synchronous Parallel (HBSP^k) [Wil00]. HCGM models parallel computers consisting of p heterogeneous processors.

Since processors have varying computing capabilities, s_i represents the speed of processor P_i. The model assumes memory and communication speeds of the processors are proportional to their computational speeds.

The HBSP^k model is an extension of the BSP model of parallel computation. In the HBSP^k model, HBSP is synonymous with HBSP^1. The HBSP^k model extends BSP hierarchically to address k-level heterogeneous parallel systems. Here, k represents the number of network layers present in the heterogeneous environment. Unlike BSP, the HBSP^k model describes multiple heterogeneous parallel systems connected by some combination of internal buses, local-area networks, campus-area networks, and wide-area networks. Furthermore, the HBSP^k incorporates parameters that reflect the relative computational and communication speeds of each of the k levels. An HBSP^k computation consists of some combination of super^i-steps. During a super^i-step, each level i node performs asynchronously some combination of local computation, message transmissions to other level i machines, and message arrivals from its peers. A message sent in one super^i-step is guaranteed to be available to the destination machine at the beginning of the next super^i-step. Each super^i-step is followed by a global synchronization of all the level i computers.

Communication models have been developed recently for improving the performance of point-to-point and collective communication operations on HNOCs. The Efficient Collective Operations package [LB96], built on top of PVM, proposes heuristics to partition the participating workstations of a collective communication operation into subnetworks based on pair-wise round-trip latencies. Next, it divides the required communication steps into two major phases: inter-subnetwork and intra-subnetwork. Different trees are used for performing collective communication operations in each of these phases. However such latency measurements do not show the impact of heterogeneity on communication send/receive overhead (the *fixed* component and the *variable* component depending on the message length). Banikazemi *et al.* [BMD98] propose a model that addresses this issue by estimating the cost of point-to-point communication on HNOCs and uses it for characterizing the performance of different collective communication operations. The model takes into account both the *fixed* component of communication send/receive overhead and the *variable* component of communication overhead and the transmission component.

The performance of the MPI's collective communications is critical in most MPI-based applications. A general algorithm (which employs a best collective communication algorithm using best buffer size with optimal number of processors involved in the collective communication) for a given collective communication operation may not give good performance on all systems due to the differences in architectures, network parameters and the storage capacity of the underlying MPI implementation. [VFD00a, VFD04b] discuss an approach in which the optimum algorithm and optimum buffer size for a given collective communication on a system is determined by conducting experiments on the system. The experiments were conducted in several phases. In the first phase, the best buffer size for a given algorithm for a given number of processors is determined by evaluating the performance of the algorithm for different buffer sizes. In the second phase, the best algorithm for a given message size is chosen by repeating the first phase with a known set of algorithms and choosing the best algorithm that gives the best result. In the third phase, the first and second phases are repeated for different numbers of processors. They present techniques to reduce the large number of experiments. [GAB+05] attempt to analyze and improve collective communication in the context of the widely deployed MPI programming paradigm by extending accepted models of point-to-point communication, such as Hockney, LogP/LogGP, and PLogP. The predictions from the models were compared with the experimentally gathered data and the findings were used to optimize the implementations of collective operations in the FT-MPI library.

Bhat *et al.* [BRP99] propose a framework, which deals with the development of efficient collective communication systems on grid-based distributed computing systems. It consists of analytical models of the heterogeneous system, scheduling algorithms for the collective communication pattern, and performance evaluation mechanisms. Kielman *et al*. [KBG00] present a performance model dealing with the optimization of MPI's collective operations on clustered wide-area systems. They use two techniques: selecting suitable communication graph shapes, and splitting messages into multiple segments that are sent in parallel over different WAN links.

The HiHCoCP model [CFM+01] (for Hierarchical Hyperclusters of Heterogeneous Processors) builds on these existing models. A hypercluster is a cluster of cluster of … of cluster of processors. It characterizes a hypercluster via parameters that reflect its tri-axial heterogeneity: the individual processors' message-processing times for the various network levels, and the latencies, link-bandwidths, and capacities of the networks at each level of the hierarchy.

Lastovetsky [Las02] uses a model of the executing network of computers, where each computer is characterized by seven parameters. These are

a) the number of processors,
b) the speed of the computer demonstrated on execution of some serial test code. This value is updated at runtime by the execution of the r**econ** statement.
c) the total number of parallel processes to run on the computer,

d) the scalability of the communication layer provided by the computer, and

e) the last three parameters determine the speed of point-to-point data transfer between processes running on the same computer as function of size of the transferred data block. The fifth parameter specifies the speed of transfer of data block of 64 bytes (measured in bytes per second),

f) the speed of transfer of data block of 64^2 bytes, and

g) the speed of transfer of data block of 64^3 bytes.

The speed of transfer of a data block of an arbitrary size is calculated by interpolation of the measured speeds.

Several research efforts have dealt with the problem of performance prediction for parallel applications executed on HNOCs, that is, predict the execution times of these applications on HNOCs. Yan *et al*. [YZS96] present a two-level model to study performance predictions for parallel computing on HNOCs. On the top level a semi-deterministic task graph is used to capture the parallel execution behavior including the variances of communications and synchronizations. On the bottom level, a discrete time model is used to quantify effects from NOW systems. An iterative process is used to determine the interactive effects between network contention and task execution. Figueira and Berman [FB96] present a model, which predicts contention effects in Host/MPP coupled heterogeneous platforms. The model provides a contention measure, the *slowdown factor*, to adjust the computation times and communication costs of an application to accommodate for system load. The adjusted applications can be used to rank candidate schedules of application tasks to system resources.

Kishimoto and Ichikawa [KI04] adopt a multiprocessing approach to estimate the best processing element (PE) configuration and process allocation based on an execution-time model of the application. The execution time is modeled from the measurement results of various configurations. Then, a derived model is used to estimate the optimal PE configuration and process allocation.

Anglano [Ang98] developed a methodology for the construction of performance models whose analysis allows the estimation of the execution time of parallel applications on HNOCs. The methodology uses Timed Petri Nets to represent the behavior of parallel programs, and a contention model based on queuing theory to quantify the effects of resource contention on the execution time of the applications processes.

Clematis and Corana [CC99] propose a performance model of heterogeneous networks of computers for analysis of the performance of heterogeneous parallel algorithms in order to predict their efficiency without real execution of the algorithms in heterogeneous environments.

Lastovetsky [Las02] uses execution time estimation models to predict the total time of execution of the algorithm on the underlying hardware without its real execution. The estimations are then used to solve the problem of selection of optimal set of processes executing on different computers of the heterogeneous network. These estimation models are based on the performance model of the parallel algorithm, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

Several authors consider scalability more important for heterogeneous parallel algorithms than efficiency.

Zhang and Yan [ZY95] present models which quantify the heterogeneity of networks and characterize the performance effects. The models consider effects of both the heterogeneity and time-sharing in a non-dedicated environment. Speedup, efficiency, and scalability are defined.

These metrics define heterogeneity based on the different loads on the participating machines. The models assumed a cluster of similar machines connected by a uniform network and each machine can execute on task (process) at a given point of time. Al-Jaroodi *et al*. [AMJ+03] extend the metrics to evaluate the performance of heterogeneous parallel applications. The extensions mainly accommodate the varying platforms and operating environments used and the possibility of having multiple tasks of the parallel application on each machine.

Donaldson *et al.* [DBP94] consider a theoretical basis for calculating speedup in a heterogeneous environment, and give definitions for speedup and superlinear speedup in a heterogeneous network. Based on these definitions, it is observed that speedup for an arbitrary task graph can be viewed as having both a heterogeneous component and a parallel component. Additional analysis of the special case of linear task graphs shows that in a heterogeneous network, not only is superlinear speedup (When adding more CPUs to parallel execution of a program, the program normally speeds up in conformity with Amdahl's Law. However, when adding CPUs accidentally relieves some other bottleneck, the speedup can exceed the number of CPUs added. This is superlinear speedup: improvement out of proportion to the hardware added) possible, but unbounded speedup is possible even without exploiting parallelism across machines.

Post and Goosen [PG01] suggest that traditional measures used for evaluating parallel performance, such as speedup and efficiency, are not appropriate for evaluating parallel performance of a heterogeneous system. They illustrate the use of linear speed to be a better alternative. Linear speed is essentially the inverse of elapsed time. They also show how linear efficiency may be used to evaluate parallel performance of a heterogeneous system, and help assess how efficient the system is. Linear efficiency is an extension to the linear speed and is calculated by dividing the potential linear speed of a system (sum of serial linear speeds of processors used) by the actual linear speed achieved by the system for a parallel execution.

Sun [Sun02] studies the relation between scalability and execution time. Based on newly uncovered relations, the concept of range comparison is introduced. Unlike conventional execution time comparison in which performance is compared at a given parallel platform and at a specified system and problem size, range comparison compares performance over a wide range of ensemble and problem size via performance crossing point analysis. Crossing point analysis finds slow/fast performance crossing points of parallel algorithms and machines.

2.4.1 Summary

The proposed models demonstrate that a small set of machine characteristics must be taken into consideration when programming high-performance computations on HNOCs: computing bandwidth, communication latency, communication overhead, communication bandwidth, network contention effects and memory hierarchy. Early computational models used a few parameters to describe the features of parallel machines. However recent models attempt to bridge the gap between software and hardware by using more parameters to capture the essential characteristics of parallel machines. A comprehensive performance model of heterogeneous networks of computers should be able to accomplish the following tasks:

- Analysis of the performance of heterogeneous parallel algorithms in order to accurately predict their efficiency without real execution of the algorithms in heterogeneous environments,
- Accurately predict the execution time of parallel applications to provide efficient mappings on HNOCs.

Such a comprehensive model is still far away from realization.

2.5 Static and Dynamic Mapping Strategies

To minimize the execution time of a parallel application running on a heterogeneous computing system, an appropriate mapping scheme to allocate the application tasks to the processors is needed. The general problem of mapping tasks to processors (defined as matching and scheduling) has been shown to be NP-complete [IK77, Fer89]. Norman and Thanisch [NT93] classify versions of the mapping problem and present existing research results. We present a brief summary of their work here for completeness.

No Task Precedence

The simplest and the most computationally tractable models of parallel computation are those where computations are modeled as tasks, each of which is executed sequentially on a single processor and between which there is no communication. The problem is simply an assignment problem, one of balancing the load on the various processors to which the computation is being mapped.

Model 1: No Precedence

An instance of the model can be formulated as follows: An instance A of the model is a 3-Tuple, a tuple is a finite sequence of objects, (P, Γ, f), where P is a set of n processors, Γ is set of l tasks and $f : \Gamma \rightarrow Z_0^+$ (where Z_0^+ is a set of positive integers including zero) is a function such that $f(\gamma)$ returns the time to compute task γ. Let F_A denote the set of all surjective mappings from Γ to the collection of singleton subsets of P. A function f from a set X to a set Y is said to be surjective, if and only if for every element y of Y , there is an element x in X such that $f(x) = y$, that is, f is surjective if and only if $f(X) = Y$. A singleton is a set containing a single element. F_A may be thought of as the set of possible task mapping functions. For each $g_m \in F_A$ let $h_m : P \rightarrow 2^{\Gamma}$ denote the function which returns the set of tasks mapped to a given processor by mapping function g_m. The **makespan** M_m associated with the mapping function g_m is defined as:

$$M_m = \max_{p \in P} \left(\sum_{\gamma \in h_m(p)} f(\gamma) \right)$$

We can pose the following decision problem:

Decision Problem 1 *Given an instance A of the above model and a positive integer k does there exist a function* $g_m \in F_A$ *such that* $M_m < k$*?*

Decision Problem 1 is NP-complete in the case of two or more identical processors.

Tasks with Precedence

Multicomputer programs with inter-task communications are better modeled by an alternative formulation. Below a model of non-preemptive scheduling is described where tasks show dependencies, and the dependency is satisfied at the termination of the precedent task.

Model 2: Precedence With No Cost

An instance A of the model is a 3-Tuple (P, Λ, f_c), where P is a set of n processors; $\Lambda = (\Gamma, \Delta)$ is a directed acyclic graph DAG (A DAG is a directed graph that contains no cycles) where Γ is a set of l tasks and Δ represents a partial order on the tasks; $f_c : \Gamma \rightarrow Z_0^+$ is a function such that $f_c(\gamma)$ returns the time to compute task γ. Let F_A denote the set of all surjective mappings from Γ to the collection of singleton subsets of P. F_A may be thought of as the set of possible task mapping functions. For each $g_m \in F_A$ let $h_m : P \rightarrow 2^{\Gamma}$ denote the function which returns the set of tasks mapped to a given processor by mapping function g_m. For each $g_m \in F_A$ let S_m denote the set of valid schedules for a given mapping g_m. The makespan M_s of a schedule $s \in S_m$ is given simply by:

$$M_s = \max_{\gamma \in \Gamma}(s(\gamma) + f_c(\gamma))$$

Decision Problem 2 *Given an instance A of the model 2 and an integer k, does there exist a mapping function $g_m \in F_A$ such that there exists a schedule $s \in S_m$ such that $M_s < k$?*

Decision Problem 2 is NP-complete for general *n*, even if the range of f_c is {1}, or in the case of $n = 2$ if the range of f_c is {1,2} (the range of a function is the set of all values produced by a function).

Task Precedence and Communication Delays

The model described before captures the essence of interprocessor communication in terms of the implied precedence, but fails to capture any of the overheads associated with message transfer. This and the following models are extensions to the previous model which attempt to characterize the overheads of communication in different ways.

Model 3: Precedence With Communication Delay

An instance *A* of the model is a 4-Tuple (P, Λ, f_c, τ), where *P* is a set of *n* processors; $\Lambda = (\Gamma, \Delta)$ is a DAG where Γ is a set of *l* tasks and Δ represents a partial order on the tasks; $f_c : \Gamma \to Z_0^+$ is a function such that $f_c(\gamma)$ returns the time to compute task γ and τ is an integer communication delay. Let F_A denote the set of all task mapping functions g_m, such that $g_m : \Gamma \to 2^P$. Function $g_m \in F_A$ returns the set of processors on which a task is executed in the mapping defined by g_m. For each g_m we define a corresponding function let $h_m : P \to 2^\Gamma$ denote the function which returns the set of tasks mapped to a given processor by mapping function g_m. For each g_m we define the set S_m of allowable schedules *s* such that $s : \Gamma \times P \to Z_0^+$ where, for any given $s \in S_m$, if $\gamma \notin h_m(p)$, then $s(\gamma,p)$ is undefined, otherwise $s(\gamma,p)$ is the time at which task γ is executed on processor *p*. The makespan M_s of a schedule $s \in S_m$ is given simply by:

$$M_s = \max_{p \in P} \max_{\gamma \in h_m(p)} (s(\gamma, p) + f_c(\gamma))$$

Decision Problem 3 *Given an integer k and a 3-tuple B=(Λ, f_c, τ), where the range of f_c is restricted to {1}, does there exist an instance A = (P, Λ, f_c, τ) of Model 3 for which there is a mapping function $g_m \in F_A$, and an associated scheduling function $s \in S_m$ such that $M_s < k$?*

Decision Problem 3 is NP-complete.

Cost Based Models

Model 4: Communication Costs and Computation Costs

An instance *A* of the model is a 4-Tuple (*P*, *Λ*, f_d, f_e), where *P* is a set of *n* processors; *Λ* = (*Γ*,*Δ*) is an undirected graph where *Γ* is a set of *l* tasks and *Δ* is a set of undirected edges corresponding to communication between processes; $f_d : \Gamma \times P \to Z_0^+$ is a function such that $f_d(\gamma, p)$ returns the time to compute task γ on processor *p*; $f_e : \Delta \to Z_0^+$ is a function returning the cost associated with communication between processes if they are mapped to different processors. Given *A*, we can consider F_A of functions which map from *Γ* onto *P*. This may be thought of as the set of all task mapping functions. For each $g_m \in F_A$ we can define a corresponding function $h_m : P \to 2^\Gamma$, which returns the set of tasks mapped to a given processor by mapping function g_m. Now, the global cost of computation associated with mapping function g_m, is given by:

$$U_m = \sum_{\gamma \in \Gamma} f_d(\gamma, g_m(\gamma))$$

and the global cost of communication associated with g_m is:

$$V_m = \sum_{\{\gamma,\delta\} \in \Delta | g_m(\gamma) \neq g_m(\delta)} f_e(\langle \gamma, \delta \rangle)$$

And the total cost R_m of a mapping is given by:

$$R_m = U_m + V_m$$

Decision Problem 3 *Given an instance A of Model 4 and an integer k: does there exist a mapping function* $g_m \in F_A$ *such that* $R_m < k$*?*

Stone [Sto77a, Sto77b, Sto78] shows an optimal algorithm for Model 4 with *n*=2. Fernandez-Baca [Fer89] has proved that Decision Problem 4 is NP-complete if all of the following restrictions hold:

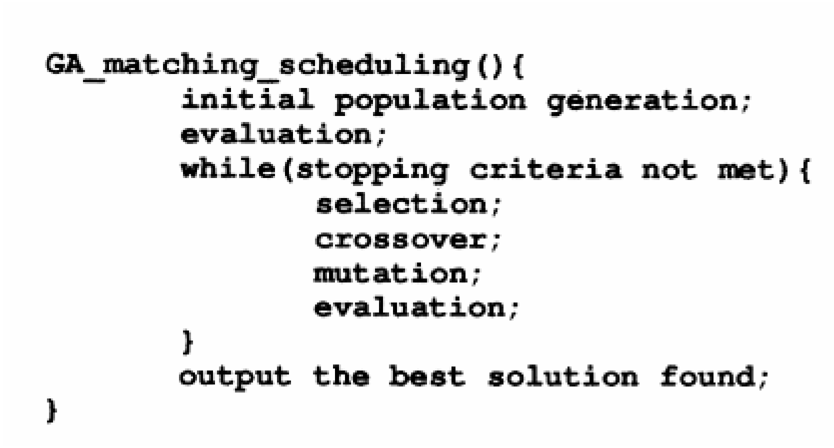
- The range of f_d is {0},
- The range of f_e is {1},
- *n*=3,
- *A* is both planar and bipartite. In graph theory, a planar graph is a graph that can be embedded in a plane so that no edges intersect and a bipartite graph is a special graph where the set of vertices can be divided into two disjoint sets with two vertices of the same set never sharing an edge.

Many heuristics (approximation algorithms) have been developed to obtain near-optimal solutions to the mapping problem. Mapping algorithms are usually classified as static or dynamic. In static mapping, mapping decisions are taken before executing the application and are not changed until the end of the application. In dynamic mapping, mapping decisions are taken while the program is running.

In [BSB+99], a collection of eleven static mapping heuristics has been studied and compared by simulation studies under one set of common assumptions. First, some preliminary terms must be defined. Let a meta-task be defined as a collection of independent tasks with no data dependencies. It is assumed that the size of the meta-task (number of tasks to execute), *t*, and the number of machines in the heterogeneous computing environment, *m*, are static and known *a priori*. It is assumed that an accurate estimate of the expected execution time for each task on each machine is known priori to execution and contained within an *ETC* (expected time to compute) matrix. *ETC*(*i*,*j*) gives the estimated execution time for task *i* on machine *j*. Machine availability time, *mat*(*j*), is the earliest time a machine *j* can complete the execution of all the tasks that have been previously assigned to it. Completion time, *ct*(*i*,*j*), is the machine availability time plus the execution time of task *i* on machine *j*, i.e., $ct(i,j) = mat(j) + ETC(i,j)$, where . The performance criterion used to compare the results of the heuristics is the maximum value of *ct*(*i*,*j*), for $0 \leq i < t$ and $0 \leq j < m$, for each mapping, also known as the makespan. Each heuristic is attempting to minimize the makespan (i.e., finish execution of the meta-task as soon as possible).

The static mapping heuristics are

- **Opportunistic Load Balancing (OLB)**. OLB assigns each task, in arbitrary order, to the next available machine, regardless of the task's expected machine time on that machine.
- **User Directed Assignment (UDA)**. UDA assigns each task, in arbitrary order, to the machine with the best expected execution time for that task, regardless of that machine's availability.
- **Fast Greedy**. Fast Greedy assigns each task, in arbitrary order, to the machine with the minimum completion time for that task.
- **Min-min**. The Min-min heuristic begins with the set of all unmapped tasks. Then, the set of minimum completion times is found. Next, the task with the overall *minimum* completion time is selected and assigned to the corresponding machine. Intuitively, Min-min attempts to map as many tasks as possible to their first choice of machine (on the basis of completion time), under the assumption that this will result in shorter makespan.



```
GA_matching_scheduling(){
        initial population generation;
        evaluation;
        while(stopping criteria not met){
                selection;
                crossover;
                mutation;
                evaluation;
        }
        output the best solution found;
}
```

Figure 2.5: General procedure for a Genetic Algorithm, based on [SP94].

- **Max-min.** The Max-min heuristic is similar to Min-min heuristic except that the task with the overall *maximum* completion time is selected and assigned to the corresponding machine. The motivation behind Max-min is to attempt to minimize the penalties incurred by delaying the scheduling of long-running tasks.
- **Greedy**. The Greedy heuristic is literally a combination of Min-min and Max-min heuristics. It performs both of the Min-min and Max-min heuristics, and uses the better solution.
- **Genetic Algorithm (GA)**. Genetic Algorithms (GAs) are a popular technique used for searching large solution spaces (e.g., [SY96, WSR+97]). Figure 2.5 shows the steps in a general Genetic Algorithm: (1) an encoding, (2) an initial population, (3) an evaluation using a particular fitness function, (4) a selection mechanism, (5) a crossover mechanism, (6) a mutation mechanism, and (7) a set of stopping criteria.

  The characteristics of this GA-based approach [WSR+97] include: separation of the matching and the scheduling representations, independence of the chromosome structure from the details of the communication subsystem, and consideration of overlap among all computations and communications that obey subtask precedence constraints. Each chromosome consists of two parts: the matching string and the scheduling string. Thus, a chromosome represents the subtask-to-machine assignments (matching) and the execution ordering of the subtasks assigned to the same machine.

The GA implemented operates on a population of 200 chromosomes (possible mappings) for a given meta-task. The initial population is generated using two methods: (a) 200 randomly generated chromosomes from a uniform distribution, or (b) one chromosome that is a Min-min solution and 199 random solutions (mappings). The GA executes eight times (four times with initial populations from each method), and the best of the eight mappings is used as the final solution.

After the generation of the initial population, all of the chromosomes in the population are evaluated (i.e., ranked) based on their fitness value (i.e., makespan), with the smaller fitness value being a better mapping. Then the main loop in Figure 2.5 is entered and a rank based roulette wheel scheme [SP94] is used for selection.

Next, the crossover operation selects a pair of chromosomes and chooses a random point in the first chromosome. After crossover, the mutation operation is performed. Mutation randomly selects a task within the chromosome, and randomly reassigns it to a new machine.

Finally, the chromosomes from this modified population are evaluated again. The GA stops when any one of three conditions is met: (a) 1000 total iterations, (b) no change in the elite chromosome for 150 iterations, and (c) all chromosomes converge.

- **Simulated Annealing (SA)**. SA is an iterative technique that considers only one possible solution (mapping) for each meta-task at a time. This solution uses the same representation for a solution as the chromosome for the GA. SA uses a procedure that probabilistically allows poorer solutions to be accepted to attempt to obtain a better search of the solution space (e.g., [CP96, KGV83, RN95]). This probability is based on a system temperature that decreases for each iteration. As the system temperature "cools," it is more difficult for currently poorer solutions to be accepted. The initial system temperature is the makespan of the initial mapping.

The specfic SA procedure implemented here is as follows. The initial mapping is generated from a uniform random distribution. The mapping is mutated in the same manner as the GA, and the new makespan is evaluated. The decision algorithm for accepting or rejecting the new mapping is based on [CP96]. If the new makespan is better, the new mapping replaces the old one. If the new makespan is worse (larger), a uniform random number $z \in [0,1)$ is selected. Then, z is compared with y, where

$$y = \frac{1}{1 + e(\frac{\text{old makespan - new makespan}}{\text{temperature}})}$$

If $z>y$ the new (poorer) mapping is accepted, otherwise it is rejected, and the old mapping is kept.

After each mutation, the system temperature is decreased by 10%. This defines one iteration of SA. The heuristic stops when there is no change in the makespan for 150 iterations or the system temperature reaches zero.

- **Genetic Simulated Annealing (GSA)**. The GSA heuristic is a combination of the GA and SA techniques. In general, GSA follows procedures similar to the GA heuristic. However, for the selection process, GSA uses the SA cooling schedule and system temperature.

- **Tabu**. Tabu search is a solution space search that keeps track of the regions in solution space, which have already been searched so as not to repeat a search near these areas.
- **A***. A* is a tree search beginning at a root node that is usually a null solution. As the tree grows, intermediate nodes represent partial solutions (a subset of tasks are assigned to machines), and leaf nodes represent final solutions (all tasks are assigned to machines).

The results obtained from executing these heuristics show that GA was the best heuristic for most cases, followed closely by Min-min. If the best mapping available in less than one minute is desired, Min-min should be used; if more time is available for finding the best mapping, GA and **A*** should be considered.

Wu and Shu [WS01] propose an algorithm, named *Relative Cost* (*RC*) algorithm, to obtain optimal mapping. The proposed algorithm retains the advantage of the Min-min algorithm and achieves good load balance at the same time. As one of its limitations, Min-min algorithm gives small tasks higher priorities and therefore assigns them early, going against the general principle that the large tasks should be mapped first for a balanced load. When small tasks execute first, it tends to execute a few larger tasks near the end, leaving some machine sitting idle, which results in poor system utilization. The *RC* algorithm uses a new criterion of *relative cost* to determine the mapping order of tasks. In the *RC* algorithm, the higher priority is given to tasks that

- have a good match between tasks and machines; and
- minimize the completion time.

Baraglia *et al.* [BFR03] propose a static graph-based mapping algorithm, called Heterogeneous Multi-Phase Mapping (*HMM*) that permits a suboptimal mapping of a parallel application onto a heterogeneous computing distributed system by using a local search technique together with a tabu search heuristic.

Maheswaran *et al*. ([MAS+99a], [MAS+99b]) study two types of dynamic mapping heuristics: on-line and batch mode heuristics. In the on-line mode, a task is mapped into a machine as soon as it arrives at the mapper. In the batch mode, tasks are not mapped onto the machines as they arrive; instead they are collected into a set that is examined for mapping at prescheduled times called mapping events. While on-line mode heuristics consider a task for mapping only once, batch mode heuristics consider a task for mapping at each mapping event until the task begins execution. Each heuristic is attempting to minimize the makespan (i.e., finish execution of the meta-task as soon as possible).

The on-line mode mapping heuristics are:

- **Minimum Completion Time (MCT)**. The MCT heuristic assigns each task to the machine that results in the task's earliest completion time. This causes some tasks to be assigned to machines that do not have the minimum execution time for them.
- **Minimum Execution Time (MET)**. The MET heuristic assigns each task to the machine that performs the task's computation in the least amount of execution time. This heuristic, in contrast to MCT, does not consider machine ready times.
- **Switching Algorithm (SA)**. The SA heuristic uses the MCT and MET heuristics in a cyclic fashion depending on the load distribution across the machines.
- **K-percent Best (KPB)**. The KPB heuristic considers only a subset of machines while mapping a task. The subset is formed by picking the ($k \times m/100$) (where m is the number of machines) best machines based on the execution times for the task, where $100/m \leq k \leq 100$. The task is assigned to a machine that provides the earliest completion time.

- **Opportunistic Load Balancing (OLB)**. OLB assigns the task to the machine that becomes ready next. If multiple machines become ready at the same time, then one machine is arbitrarily chosen.

The batch mode mapping heuristics use two interval strategies. The *regular time interval* strategy maps the meta-tasks at regular intervals of time except when all the machines are busy. In the fixed count strategy, the length of the mapping intervals will depend on the arrival rate and the completion rate. The batch mode mapping heuristics studied are:

- **Min-min** and **Max-Min**. These heuristic are similar to the static mapping heuristics discussed above.
- **Sufferage**. The sufferage heuristic is based on the idea that better mappings can be generated by assigning a machine to a task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it.

Batch mode heuristics can cause some tasks to be starved of machines. To reduce starvation, ageing schemes are implemented.

In the on-line mode, the KPB heuristic outperformed the other heuristics on all performance metrics. In the batch mode, the Min-min heuristic outperformed the Sufferage and Max-min heuristics in the average sharing penalty. However, the Sufferage heuristic performed the best with respect to makespan.

Maheswaran and Siegel [MS98a] present a dynamic mapping heuristic called the hybrid remapper. The hybrid remapper is based on a centralized policy and improves a statically obtained initial mapping and scheduling by remapping to reduce the overall execution time. During application execution, the hybrid remapper uses run-time values for the subtask completion times and machine availability times whenever possible. The hybrid remapper assumes a fully connected, contention-free communication model.

Tan *et al.* [TSA+97] present a mathematical framework that models the matching of subtasks to machines, scheduling of subtasks' computation, scheduling of intermachine communication steps, and selection of sources of shared data items for intermachine communication (data relocation). Initially, it is assumed at any instant of time, only one machine is being used for program execution and only one subtask is being executed. Based on this assumption, a polynomial algorithm is introduced to optimize scheduling and data relocation with respect to any given matching of subtasks to machines. It is assumed that matching is static and has already been done.

Cierniak *et al.* [CLZ97] propose compile-time techniques for scheduling parallel loops for a HNOCs. They propose a simple model for use in compiling for a network of processors, and develop compiler algorithms for generating optimal and near-optimal schedules of loops for load balancing, communication optimizations, network contention, and memory heterogeneity.

All the static and dynamic strategies discussed above make very little suggestions about the nature of scheduled tasks (if any) considering them as a set of independent equal units. They pay more attention to the model of the heterogeneous hardware. Also it is assumed that the application program is decomposed into subtasks, each of which is computationally homogeneous. But this process of decomposition is itself a tedious and error-prone task.

Lastovetsky [Las02] present a mapping algorithm that solves the problem of optimal mapping of processes into the set of processes executing on different computers of the heterogeneous network. The solution to the problem is based on the following:

- The mpC model of the parallel algorithm, which should be executed. This model allows the programmer to define at runtime all the main features of parallel algorithm, which have an impact on the execution performance of the application on heterogeneous platforms.
- The performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.
- A map of processes of the parallel program, for each computer displaying both the total number of running processes and the number of free processes.

Each particular mapping is characterized by the estimation of the time of execution of the algorithm on the network of computers. The estimation is calculated based on the performance model of the parallel algorithm and the model of the executing network of computers. The mpC runtime system finds a mapping at runtime using an approximate solution obtained in a reasonable amount of time.

2.5.1 Summary

A heterogeneous computing system provides a variety of different machines executing an application whose subtasks have diverse execution requirements. The subtasks must be assigned to machines (matching) and ordered for execution (scheduling) such that the overall application execution time is minimized. It is well known that such a matching and scheduling (mapping) problem is, in general, NP-complete. Therefore, many heuristics have been developed to obtain near-optimal solutions to the mapping problems. The heuristics can be static or dynamic. The heuristics depend on the performance models of the computers in the executing heterogeneous network discussed previously. The quality of these heuristics depends on how accurately the performance models of the computers in the executing heterogeneous network estimate the subtask computation times on various machines and inter-machine data transfer times.

2.6 High-Performance Computing on Global Networks

A few approaches to high-performance computing on global networks have been proposed. The main software challenges include achieving high performance via parallelism, managing and exploiting component heterogeneity, resource management, file and data access, fault-tolerance, ease-of-use and user interfaces, protection and authentication, and exploitation of high-performance protocols.

NetSolve [CD96] is a system used to support high-performance scientific computations on global networks. NetSolve offers the ability to look for computational resources on a network, choose the best available, solve the problem, and return the solution to the user. Good performance is ensured by a load-balancing policy that enables NetSolve to use the computational resources as efficiently as possible. NetSolve is designed to run on any heterogeneous network and is implemented as a fault-tolerant client-server application.

Computational Grid [FK98] is a platform for the implementation of high-performance applications using widely dispersed computational resources. NASA's Information Power Grid (IPG) [JGN99] is a high-performance computing and data grid. Grid users can access widely distributed heterogeneous resources from any location, with IPG middleware adding security, uniformity, and control.

Legion [GW97] is an object-based, meta-systems software project, which connects networks, workstations, supercomputers, and other computer resources together into a system that can encompass different architectures, operating systems, and physical locations. Legion provides a coherent framework in which these elements can be combined into a metasystem. One can draw on these combined resources to parallelize complex problems and run programs more efficiently without worrying about different languages, conflicting platforms, or hardware failure. Legion seamlessly schedules and distributes your processes on available and appropriate hosts, then returns the results.

The goal of the Condor Project [LLM88] is to develop, implement, deploy and evaluate mechanisms and policies that support High Throughput Computing (HTC) on large collections of distributively owned computing resources. Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Users submit their serial or parallel jobs to Condor, Condor places them into a queue, chooses when and where to run the jobs based upon a policy, carefully monitors their progress, and ultimately informs the user upon completion. Condor's periodic checkpointing provides fault tolerance.

The core part of any software system for high-performance computing on global networks is a tool for monitoring the network performance of a global computing network. The Network Weather Service (NWS) [WS97] is such a tool that periodically monitors and dynamically forecasts the performance various network and computational resources can deliver over a given time interval. The service operates a distributed set of performance sensors (network monitors, CPU monitors, etc.) from which it gathers readings of the instantaneous conditions. It then uses numerical models to generate forecasts of what the conditions will be for a given time frame. NWS is used by dynamic schedulers and to provide statistical Quality-of-Service readings in a networked computational environment. The forecaster module in NWS applies a set of forecasting models and dynamically chooses the forecasting technique that has been most accurate over the recent set of measurements. The forecaster process in NWS produces a predicted value of deliverable performance during a specified time frame for a specified resource.

Gloperf [ASW+98] is a network performance monitoring system for grid computations built as a part of the Globus grid computing toolkit [FK97]. Globus is used to develop the fundamental technology that is needed to build computational grids, execution environments that enable an application to integrate geographically-distributed instruments, displays, and computational and information resources. Gloperf makes simple, end-to-end TCP measurements requiring no special host permissions. Gloperf is primarily a sensor and collection mechanism; it does not contain any prediction models.

The AppLeS project [BWF+96] is developing scheduling protocols from an applications point of view to provide a mechanism for scheduling individual applications at machine speeds on production heterogeneous systems. AppLeS agents utilize a network performance monitoring system such as NWS or Gloperf to monitor the varying performance of resources potentially usable by their applications. Each AppLeS uses static and dynamic application and system information to select viable resource configurations and evaluate their potential performance. AppLeS then interacts with the relevant resource management system to implement application tasks.

SmartNet [FGA+98] is a resource scheduling system for distribution environments. It allows users to execute jobs on complex networks of different computers as if they were a single machine, or meta-computer. In general, optimal multiprocessor scheduling is NP-complete, and hence SmartNet employs heuristics when it searches for a near-optimal mapping of jobs to machines and job execution schedule.

The Management System for Heterogeneous Networks (MSHN) [HKJ+99] is a resource management system for use in heterogeneous environments. Its main goal is to determine the best way to support the execution of many different applications, each with its own quality of service (*QoS*) requirements in a distributed, heterogeneous environment. MSHN evolved in part from the scheduling framework SmartNet but its research expanded into the following areas relevant to most resource management systems (RMS).

- An RMS needs to consider that the overhead of jobs sharing resources, such as networks and file servers, can have significant impact on mapping and scheduling decisions.
- An RMS must support adaptive applications.
- An RMS must deliver good QoS to many different sets of simultaneous users, some of whom may be executing compute-intensive jobs and some of whom jobs with real-time requirements.

2.7 Summary

The tools designed for programming high-performance computations on HNOCs must provide mechanisms to automate the tedious and error-prone tasks:

- Parameter determination characterizing the computational requirements of the parallel application and the capabilities of the machines,
- Data partitioning,
- Matching and Scheduling, and
- Task execution.

Ideally a tool must supply mechanisms to the programmer so that he or she can provide information to it that can assist in finding the most efficient implementation on HNOCs. Combining the system's detailed analysis with the programmer's high-level knowledge of the application is essential in finding more efficient mappings than either one alone is capable of achieving. The performance models used by the tools must take into account all the essential features underlying applications run on HNOCs, mainly, the speeds of the processors, the effects of paging and the latency and the bandwidth of the communication links between the processors. The model of the executing network of computers must take into consideration the essential set of machine characteristics such as computing bandwidth, communication latency, communication overhead, communication bandwidth, network contention effects and the memory hierarchy. Such a model must have enough parameters for it to be efficient and accurate.

We present, in the chapters to follow, tools that automate the main steps involved in application development on HNOCs. These tools employ performance model of the executing network of computers that takes into account the main features that have an essential impact on application execution performance on HNOCs.

CHAPTER 3

Outline of HMPI

This chapter presents a small set of extensions to MPI that can be used for

- Determination of the characterization parameters relevant to the computational requirements of the applications and the machine capabilities of the heterogeneous system, and
- Selection of the optimal set of processes running on different computers of the heterogeneous network.

The standard Message Passing Interface (MPI) specification provides communicator and group constructors, which allow the application programmers to create a group of processes that execute together some parallel computations to solve a logical unit of a parallel algorithm. The participating processes in the group are explicitly chosen from an ordered set of processes. This approach to the group creation is quite acceptable if the MPI application runs on homogeneous distributed-memory computer systems, one process per processor. In this case, the explicitly created group will execute the parallel algorithm typically with the same execution time as any other group with the same number of processes, because the processors have the same computing power, and the latency and the bandwidth of communication links between different pairs of processors are the same. However on HNOCs, a group of processes optimally selected by taking into account the speeds of the processors, and the latencies and the bandwidths of the communication links between them, will execute the parallel algorithm faster than any other group of processes. Selection of processes in such a group is usually a very difficult task. It requires the programmers to write a lot of complex code to detect the actual speeds of the processors and the latencies of the communication links between them, and then to use this information to select the optimal set of processes running on different computers of heterogeneous network.

The main idea of HMPI is to automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group.

The first step in this process of automation is the specification of the performance model of the heterogeneous parallel algorithm in performance model definition language. Performance model is a tool supplied to the programmer to specify his or her high-level knowledge of the application that can assist in finding the most efficient implementation on HNOCs.

The second step involves the writing of an HMPI application. A typical HMPI application consists of HMPI group management operations and the execution of the computations and communications involved in the execution of the parallel algorithm employed in the application by the members of the group. During the creation of a group of processes, HMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network. The solution to the problem is based on the following:

- The performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model.
- The performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

The accuracy of the performance model of the executing network of computers depends upon the accuracy of the estimation of the actual speeds of processors and the communication model of the executing network of computers. HMPI provides operations to dynamically update the estimation of processor speeds and parameters of communication model at runtime.

Thus if the performance model of the parallel algorithm embodies the programmer's high level knowledge of the application, the performance model of the executing network of computers expresses the detailed analysis of the executing network of computers. Using these two models is essential in finding more efficient mappings than either one alone is capable of achieving.

The main contributions in this chapter are:

a) The design of HMPI API. The main goal of the design of the API in HMPI is to smoothly and naturally extend the MPI model for heterogeneous networks of computers. This involves the design of a layer above MPI that does not involve any changes to the existing MPI API. The HMPI API must be easy-to-use and suitable for most scientific applications. The HMPI API must also facilitate transformation of MPI applications to HMPI applications that run efficiently on HNOCs.

b) The first research implementation of HMPI.

c) The design and application of HMPI+ScaLAPACK tool to speed up ScaLAPACK applications on heterogeneous networks of computers.

While presenting the HMPI API, we present additional background material just to make this chapter self-contained. This material is mainly the mapping algorithms used to solve the problem of selection of processes, the estimation procedure to estimate the time of execution for a particular mapping, and the model of a heterogeneous network of computers.

This chapter is structured as follows:

- Section 3.1 presents the specification of the performance model definition language.
- Section 3.2 presents the HMPI group management operations.

- Section 3.3 presents the HMPI operation **`HMPI_Recon`** that enables the user to dynamically update the estimation of processor speeds at runtime. This operation facilitates writing parallel programs sensitive to dynamic changing loads.
- Section 3.4 presents the HMPI operation **`HMPI_Timeof`** that allows the user to predict the total time of the algorithm execution on the underlying hardware without its real execution. This feature allows the programmer to write such a parallel program that can follow different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance.
- Section 3.5 presents the HMPI group constructor operation **`HMPI_Group_auto_create`** that detects the optimal number of processes that can execute the parallel application.
- Section 3.6 outlines the typical steps involved in the development of an HMPI program.
- Section 3.7 presents the steps involved in the transformation of an MPI program to an HMPI program.
- Section 3.8 gives an overview of a research implementation of HMPI.
- Section 3.9 summarizes the features of HMPI.

3.1 Outline of Performance Model Definition Language

HMPI allows application programmers to describe a performance model of their implemented heterogeneous algorithm. This model allows specification of all the main features of the underlying parallel algorithm that have an essential impact on application execution performance on HNOCs. These features are:

- The total number of processes executing the algorithm.

- The total volume of computations to be performed by each of the processes in the group during the execution of the algorithm,
- The total volume of data to be transferred between each pair of processes in the group during the execution of the algorithm, and
- The order of execution of the computations and communications by the involved parallel processes in the group, that is, how exactly the processes interact during the execution of the algorithm.

HMPI provides a small and dedicated model definition language for specifying this performance model. This language uses most of the features in the specification of network types of the mpC language presented in [AKL+99, LAK+00, Las02].

mpC is a high-level parallel language (an extension of ANSI C), designed specially to develop portable adaptable applications for heterogeneous networks of computers. mpC allows the programmer to define at runtime all the main features of the underlying parallel algorithm, which have an impact on the application execution performance, namely, the total number of participating parallel processes, the total volume of computations to be performed by each of the processes, the total volume of data be transferred between each pair of processes, and how exactly the processes interact during the execution of the algorithm. Such an abstraction of parallel algorithm is called a *network type* in mpC. Given a network type, the programmer can define a network object of this type and describe in details all the computations and communications to be performed on the network object.

HMPI's performance model definition language only uses the specification of the network types in mpC. The specification of performance model in HMPI is the same as the specification of the performance model in mpC in the form of network type. Thus it can be said that HMPI's model definition language is a subset of mpC language in that it uses only the feature of network types in mpC.

A compiler compiles the description of this performance model to generate a set of functions. The functions make up an algorithm-specific part of the HMPI runtime system.

We illustrate the features of the HMPI's performance model definition language with a tool, which automatically transforms ScaLAPACK [CDD+96] programs solving dense linear algebra problems on *massively parallel processors* (MPP) into programs solving the same problems on HNOCs with good performance improvements.

3.1.1 Homogeneous Distribution of Data with Heterogeneous Distribution of Processes

In this section we present a tool that transforms ScaLAPACK programs that solve dense linear algebra problems on *massively parallel processors* (MPP) into parallel applications that solve the same problems on HNOCs with good performance improvements.

The input to the tool is a homogeneous parallel algorithm that solves the problem on MPPs. The transformed application for HNOCs distributes data across parallel processes exactly in the same fashion as its homogeneous prototype. However, the transformed application uses a modified algorithm that allows more than one process involved in its execution to be run on each processor so that the number of processes running on the processor is proportional to its speed. In other words, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the heterogeneous network, and this way each processor performs the volume of computations proportional to its speed.

The strategy employed by the tool is based on HeHo strategy (heterogeneous distribution of processes over processors and homogeneous block distribution of data over the processes) presented by Kalinov and Lastovetsky [KL01]. They analyze two strategies, which are HeHo and HoHe (homogeneous distribution of processes over processors with each process running on a separate processor and heterogeneous block cyclic distribution of data over the processes). Both strategies were implemented in the mpC language. The first strategy is implemented using calls to ScaLAPACK; the second strategy is implemented with calls to LAPACK [ABB+92] and BLAS [DCD+90]. They compare the strategies using Cholesky factorization on a network of workstations. They show that for heterogeneous parallel environments both the strategies HeHo and HoHe are more efficient that the traditional homogeneous strategy HoHo (homogeneous distribution of processes over processors and homogeneous distribution of data over the processes as implemented in ScaLAPACK). They also show that HoHe strategy is more efficient than the HeHo strategy (speedup of 40% observed on networks where the ratio of the speed of the fastest processor to the speed of the slowest processor is 7.1).

The main disadvantage of the HoHe strategy is non-Cartesian nature of the data distribution (Cartesian data distribution is shown in Figure 3.16). This leads to additional communications that can be essential in the case of large networks. The HeHo strategy is easy to accomplish. It allows the usage of high-quality software, such as ScaLAPACK, developed for homogeneous distributed memory systems in heterogeneous environments and to obtain a good speedup with minimal expenses. However the HeHo strategy does not take into account processor memory size. For an application dealing with big matrices, it can cause paging, which in turn causes slowing down of the parallel application. Therefore to use this strategy it is necessary to restrict the number of processes running on processors in accordance with the estimated size of the application and the main memory available. We aim to do further research to find the crossover point between HoHe and HeHo strategies in case of large networks.

Thus the main purpose of the tool is to allow the application programmers to convert conventional parallel applications that are designed to run on MPPs to applications that run efficiently on HNOCs without rewriting these applications. The tool adopts a multiprocessing approach, which does not aim to extract the maximum performance from a heterogeneous network but provides an easy and simple way to execute a wide range of conventional applications on HNOCs with good performance improvements.

We illustrate the features of the tool by transforming a sample ScaLAPACK [CDD+96] program. ScaLAPACK is a well-known standard package of high-performance linear algebra routines for distributed-memory message passing MIMD computers and networks of workstations supporting PVM and/or MPI . It is a continuation of the LAPACK project, which designed and produced analogous software for workstations, vector supercomputers, and shared-memory parallel computers. Both libraries contain routines for solving systems of linear equations, least squares problems, and eigenvalue problems.

Consider the ScaLAPACK program computing matrix multiplication using the routine **`PDGEMM`** shown in Figure 3.1. This routine performs any of the following matrix-matrix operations:

$$C = \alpha\times A\times B+\beta\times C$$
$$C = \alpha\times A\times B^T+\beta\times C$$
$$C = \alpha\times A^T\times B+\beta\times C$$
$$C = \alpha\times A^T\times B^T+\beta\times C$$

where α and β are scalars and A, B, and C are matrices.

```
   PROGRAM HPDGEMM

   INTEGER            DLEN_, M, K, N, NB, ICTXT, INFO, MYCOL, MYROW,
   $                  NPCOL, NPROW, MP, KP, KQ, NQ,
   $                  DESCA( DLEN_ ), DESCB( DLEN_ ), DESCC( DLEN_ )
   PARAMETER          ( DLEN_ = 9, LLD_ = 9 )
   EXTERNAL           BLACS_EXIT, BLACS_GRIDEXIT, BLACS_GRIDINFO,
   $                  DESCINIT, PDGEMM
*
*  Define process grid
*
   CALL BLACS_GET( -1, 0, ICTXT )
   CALL BLACS_GRIDINIT( ICTXT, ‘Row-major’, NPROW, NPCOL )
   CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )

   MP = NUMROC( M, NB, MYROW, 0, NPROW )
   KP = NUMROC( K, NB, MYROW, 0, NPROW )
   KQ = NUMROC( K, NB, MYCOL, 0, NPCOL )
   NQ = NUMROC( N, NB, MYCOL, 0, NPCOL )
*
*  Initialize the array descriptors for the matrices A, B, and C
*
   CALL DESCINIT( DESCA, M, K, NB, NB, 0, 0, ICTXT, MAX( 1, MP ),
   $              INFO )
   CALL DESCINIT( DESCB, K, N, NB, NB, 0, 0, ICTXT, MAX( 1, KP ),
   $              INFO )
   CALL DESCINIT( DESCC, M, N, NB, NB, 0, 0, ICTXT, MAX( 1, MP ),
   $              INFO )
*
*  Generate random matrices A, B, and C
*
   CALL PDMATGEN( … )
*
*  Call the ScaLAPACK routine PDGEMM
*
   CALL PDGEMM( ‘No transpose’, ‘No transpose’, M, N, K, 1., A, 1, 1,
        DESCA, B, 1, 1, DESCB, 1., C, 1, 1, DESCC )
*
*  Release the process grid
*  Free the BLACS context
*
   CALL BLACS_GRIDEXIT( ICTXT )
*
*  Exit the BLACS
*
   CALL BLACS_EXIT( 0 )
*
   STOP
   END
```

Figure 3.1: The most relevant fragments of code of the ScaLAPACK program computing matrix-matrix multiplication using **`PDGEMM`**.

```
int main(int argc, char **argv) {
     static int p, q, m, n, k, kb, kp, kq, mycol, myrow, ictxt, desca[9], input_p[2],
               descb[9], descc[9], rc, info, i, mp, mq, info, i__1, nrhs,
               iaseed, ibseed, icseed, c__0 = 0, c__1 = 1, nd, **dp, output_p;
     static double *a, *b, *c;
     void *model_params;
     HMPI_Group gid;
     HMPI_Init(argc, argv);
     // Estimation of speeds of the processors
     if (HMPI_Is_member(HMPI_PROC_WORLD_GROUP)
        HMPI_Recon(&fdgemm, input_p, 2, &output_p);
     // Model parameter initialization
     if (HMPI_Is_host()) {
        model_params[0] = n;
        model_params[1] = kb;
     }
     // HMPI Group creation
     if (HMPI_Is_host())
        HMPI_Group_auto_create(&gid, &HMPI_Model_pdgemm, model_params);
     if (HMPI_Is_free())
        HMPI_Group_auto_create(&gid, &HMPI_Model_pdgemm, NULL);
     // Execution of the algorithm
     if (HMPI_Is_member(&gid)) {
        MPI_Comm algocomm = *(MPI_Comm*)HMPI_Get_comm(&gid);
        HMPI_Group_topology(&gid, &nd, dp);
        p = (*dp)[0];
        q = (*dp)[1];
        ictxt = Csys2blacs_handle(algocomm);
        // Form BLACS context based on algocomm
        Cblacs_gridinit(&ictxt, "r", p, q);
        // Initialize the process grid
        blacs_gridinfo__(&ictxt, &p, &q, &myrow, &mycol);
        mp = numroc_(&m, &kb, &myrow, &c__0, &p);
        kp = numroc_(&k, &kb, &myrow, &c__0, &p);
        kq = numroc_(&k, &kb, &mycol, &c__0, &q);
        nq = numroc_(&n, &kb, &mycol, &c__0, &q);
        i__1 = max(1,mp);
        descinit_(desca, &m, &k, &nb, &nb, &c__0, &c__0, &ictxt, &i__1, &info);
        i__1 = max(1,kp);
        descinit_(descb, &k, &n, &nb, &nb, &c__0, &c__0, &ictxt, &i__1, &info);
        i__1 = max(1,mp);
        descinit_(descc, &m, &n, &nb, &nb, &c__0, &c__0, &ictxt, &i__1, &info);
        iaseed = 100;
        pdmatgen_(&ictxt, "No transpose", "No transpose", &desca[2],
              &desca[3], &desca[4], &desca[5], a, &desca[8], &desca[6],
              &desca[7], &iaseed, &c__0, &mp, &c__0, &kq, &myrow, &mycol, &p, &q);
        ibseed = 200;
        pdmatgen_(&ictxt, "No transpose", "No transpose", &descb[2],
              &descb[3], &descb[4], &descb[5], b, &descb[8], &descb[6],
              &descb[7], &ibseed, &c__0, &kp, &c__0, &nq, &myrow, &mycol, &p, &q);
        // Compute C=A×B
        pdgemm_( &ictxt, "No transpose", "No transpose", &descc[2], &descc[3],
        &descc[4], &descc[5], c, &descc[8], &descc[6], &descc[7], &icseed, &c__0, &mp,
        &c__0, &nq, &myrow, &mycol, &p, &q);
        // Release the process grid, Free the BLACS context
        blacs_gridexit__(&ictxt);
     }
     // HMPI Group Destruction
     if (HMPI_Is_member(&gid))
        HMPI_Group_free(&gid);
     HMPI_Finalize(0);
}
```

Figure 3.2: The most relevant fragments of generated HMPI code computing matrix-matrix multiplication using **PDGEMM** on heterogeneous networks.

There are four basic steps involved in calling a ScaLAPACK routine.

1. Initialize the process grid. The **`BLACS_GET`** routine returns the default system context for input to **`BLACS_GRIDINIT`**. The routine **`BLACS_GRIDINIT`** is called to map the processes sequentially in row-major order into the process grid. The first parameter to this routine is the system context to be used in creating the BLACS context. The second parameter to this routine indicates how to map processes into the process grid. The third and fourth parameters indicate the number of rows and number of columns in the process grid. The routine **`BLACS_GRIDINFO`** returns the row and column index in the process grid of the calling process.
2. Distribution of the matrix on the process grid. Each global matrix that is to be distributed across the process grid must be assigned an array descriptor using the ScaLAPACK TOOLS routine **`DESCINIT`**. A mapping of the global matrix onto the process grid is accomplished using the user-defined routine `PDMATGEN`.
3. Call the ScaLAPACK routine **`PDGEMM`**.
4. Release the process grid via a call to **`BLACS_GRIDEXIT`**. When all the computations have been completed, the program is exited with a call to **`BLACS_EXIT`**.

3.1.2 Main Constructs of Performance Model Definition Language

This program is input to the tool, which generates a C program shown in Figure 3.2. The tool uses the performance model definition **`pdgemm`** shown below:

```
/* 1 */ algorithm pdgemm(int n, int b, int p, int q)
/* 2 */ {
/* 3 */   coord I=p, J=q;
/* 4 */   node {I>=0 && J>=0: bench*((n/(b*p))*(n/(b*q))*(n/b));};
/* 5 */   link (K=p, L=q)
/* 6 */   {
```

```
/* 7 */      I>=0 && J>=0 && I!=K :
/* 8 */        length*((n/(b*p))*(n/(b*q))*(b*b)*sizeof(double))
/* 9 */              [I, J]->[K, J];
/* 10 */     I>=0 && J>=0 && J!=L:
/* 11 */       length*((n/(b*p))*(n/(b*q))*(b*b)*sizeof(double))
/* 12 */             [I, J]->[I, L];
/* 13 */   };
/* 14 */   parent[0,0];
/* 15 */   scheme
/* 16 */   {
/* 17 */     int i, j, k;
/* 18 */     for(k = 0; k < n; k+=b)
/* 19 */     {
/* 20 */       par(i = 0; i < p; i++)
/* 21 */          par(j = 0; j < q; j++)
/* 22 */             if (j != ((k/b)%q))
/* 23 */               (100.0/(n/(b*q))) %% [i,((k/b)%q)]->[i,j];
/* 24 */       par(i = 0; i < p; i++)
/* 25 */          par(j = 0; j < q; j++)
/* 26 */             if (i != ((k/b)%p))
/* 27 */               (100.0/(n/(b*p))) %% [((k/b)%p),j]->[i,j];
/* 28 */       par(i = 0; i < p; i++)
/* 29 */         par(j = 0; j < q; i++)
/* 30 */           ((100.0×b)/n) %% [i,j];
/* 31 */     }
/* 32 */   };
/* 33 */ };
```

This performance model definition describes the simplest scenario performed by the **pdgemm** routine in ScaLAPACK, which uses outer-product algorithm using the *logical LCM hybrid algorithmic blocking* strategy [PD99]. The performance model definition describes the parallel matrix-matrix multiplication of two dense square matrices *A* and *B* of size **n×n**. The distribution blocking factor **b** used in the matrix-matrix multiplication is assumed to be equal to the algorithmic blocking factor. The performance model definition also assumes that the matrices are divided into whole number of blocks of size equal to distribution blocking factor, that is, `(n%(b×p))` and `(n%(b×q))` (see explanation of variables below) are both equal to zero.

<u>Coordinate Declaration</u>

Line 1 is a header of the performance model declaration. It introduces the name of the performance model **`pdgemm`** parameterized with the scalar integer parameters **`n`**, **`b`**, **`p`**, and **`q`**. Parameter **`n`** is the size of square matrices *A*, *B*, and *C*. Parameter **`b`** is the size of the distribution blocking factor. Parameters **`p`** and **`q`** are output parameters representing the number of processes along the row and the column in the process grid arrangement. The scope of the parameters is the corresponding performance model declaration. The declaration of the performance model is also called a *topology*.

The body of the performance model declaration starts at line 3. Line 3 is a *coordinate declaration* declaring the coordinate system to which the processor nodes of the network are related. It introduces integer *coordinate variable* **I** ranging from **`0`** to **`p-1`**, and integer *coordinate variable* **J** ranging from **`0`** to **`q-1`**. For example, the coordinate declaration

```
coord x = 100, y = 10, z = p;
```

declares a 3-D coordinate system, which a network containing up to **`100×10×p`** nodes may be related to.

Node Declaration

Line 4 is a *node declaration*. It relates the virtual processors to the coordinate system declared and specifies the (absolute) volume of computations to be performed by each of the processors. Line 4 declares that the relative volume of computations to be performed by the virtual processor with coordinates **(I,J)** is **`((n/(b*p))*(n/(b*q))*(n/b))`**. Line 4 also stands for the predicate ***for all*** `0≤I<p and 0≤J<q` ***then*** *a virtual processor, whose relative volume of computations is `((n/(b*p))*(n/(b*q))*(n/b))`, is related to the point with coordinate `[I,J]`.*

Consider the following performance model definition

```
algorithm Web(int m, int n, int d[m][n]) {
  coord I=m, J=n;
  node {
    I==0 && J>0: void;
    I==0 && J==0: d[I][J];
    default: d[I][J]*n;
  };
  parent [0, 0];
 };
```

In the node declaration, a processor node of the type **void** has no data and does not take part in computations. The equivalent interpretation is that the type **void** indicates that no processor is related to the positions with the corresponding coordinates. In this example, the keyword **void** in the position of the processor type indicates that no processors are related to the points with coordinates `[0,J]`, where $(0 \leq J < n)$.

When processing a node declarator, the compiler evaluates the (logical) expression for every permissible set of values of the coordinate variables. If the value is non-zero (that corresponds to the logical value **true**), a processor of the specified type and performing the specified volume of computations is related to the coordinates.

The **default** node declarator declares the volume of computations performed by all the processor nodes whose coordinates do not satisfy any (logical) expression in the rest of the node declarators of the node declaration. If there does not exist a default node declarator, these processor nodes shall have the type **void**.

Therefore in this example, the virtual processor with coordinates `[0,0]` performs relative volume of computations equal to **`d[0][0]`** whereas the rest of the virtual processors, that is processors with coordinates `[I,J]`, where $(0 < I < m)$ and $(0 \leq J < n)$, perform relative volume of computations equal to **`d[I][J]*n`**.

Specification of the volume of computation is not as easy. What is the natural unit of computation to measure the volume of computations performed by the process? The main requirement is that if given the volume of computation measured in those units, the HMPI runtime system should be able to accurately estimate the time of execution of the corresponding computations by any process of the program.

The solution proposed in the performance model definition language is that the very code that was used to estimate the speed of physical processors of the executing network can also serve as a unit of measure for the volume of computation performed by processes of the parallel algorithm.

The line 4 of node declaration specifies that the volume of computations to be performed by the virtual processor with coordinates **(I,J)** is **`((n/(b*p))*(n/(b*q))*(n/b))`** times bigger than the volume of computations performed by the benchmark code. The statement **`bench`** just specifies that as a unit of measurement, the volume of computation performed by some benchmark code is used. It is presumed that the benchmark code, which is used for estimation of speed of physical processors, multiplies two dense square **b**×**b** matrices.

Link Declaration

Lines 5-13 are a *link declaration*. This specifies:

- the links between the virtual processors,
- the pattern of communication among the abstract processors, meaning a set of communication links over which the abstract processors communicate with each other, and

- the total volume of data to be transferred between each pair of virtual processors during the execution of the algorithm.

Lines 7-9 of the link declaration describe communications related to matrix *A*. Obviously, abstract processors from the same column of the processor grid do not send each other elements of matrix *A*. Only abstract processors from the same row of the processor grid send each other elements of matrix *A*. Abstract processor P_{IJ} will send **`(n/(b×p))×(n/(b×q))×b×b`** number of elements of matrix *A* to processor P_{KJ}. The volume of data in one **`b`**×**`b`** block is given by **`(b*b)*sizeof(double)`** and so the total volume of data transferred from processor P_{IJ} to processor P_{KJ} will be given by **`(n/(b×p))×(n/(b×q))×b×b×sizeof(double)`**.

Lines 7-9 also stand for the predicate ***for all*** `0≤I<p and 0≤J<q` ***if*** `I≠K` ***then*** *there exists a link connecting virtual processors with coordinates [`I,J`] and [`K,J`] and the total amount of data transferred through this link from [`I,J`] to [`K,J`] is `(n/(b*p))*(n/(b*q))*(b*b)*sizeof(double) during the execution of the algorithm`.*

The second statement in the **`link`** declaration describes communications related to matrix *B*. Obviously, only abstract processors from the same column of the processor grid send each other elements of matrix *B*. In particular, processor P_{IJ} will send all its **`b`**×**`b`** blocks of matrix *B* to all other processors from column *J* of the processor grid. Abstract processor P_{IJ} will send **`(n/(b×p))×(n/(b×q))×b×b`** number of elements of matrix *B* to processor P_{IL}. The volume of data in one **`b`**×**`b`** block is given by **`(b*b)*sizeof(double)`** and so the total volume of data transferred from processor P_{IJ} to processor P_{IL} will be given by **`(n/(b×p))×(n/(b×q))×b×b×sizeof(double)`**.

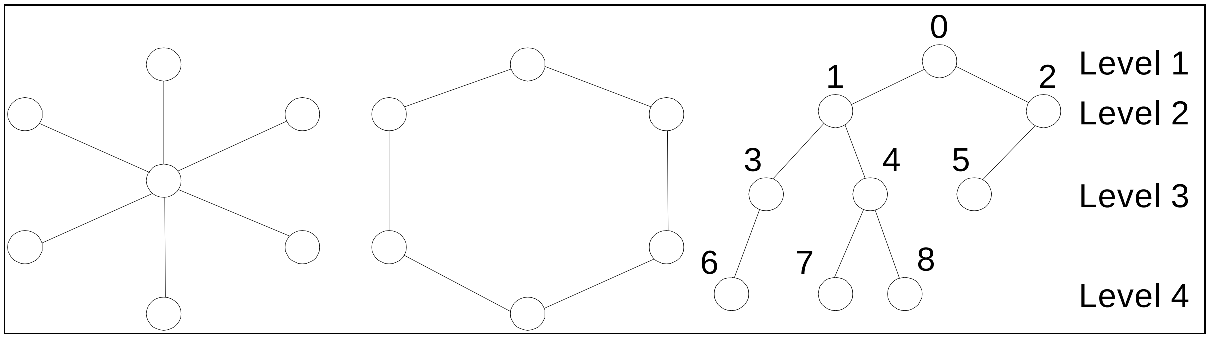


Figure 3.3: The pattern of communication among the processors. (a) A star communication pattern, (b) A ring communication pattern, and (c) A tree communication pattern.

In general, the performance model definition language allows static and dynamic communication patterns.

The communication pattern shown in Figure 3.3(a) with the performance model definition shown below represents *star* pattern.

```
algorithm Star(int p, int comm) {
  coord I=p;
  link {
    I>0: length*(comm*sizeof(double)) [I]->[0];
  };
  …
};
```

The communication pattern shown in Figure 3.3(b) with the performance model definition shown below represents *ring* pattern.

```
algorithm Ring(int p, int comm) {
  coord I=p;
  link {
    I>=0: length*(comm*sizeof(double)) [I]->[(I+1)%p];
  };
  …
};
```

The communication pattern shown in Figure 3.3(c) with the performance model definition shown below represents *tree* pattern.

```
algorithm Tree(int p, int comm) {
  coord I=p;
  link {
    I>=0: length*(comm*sizeof(double)) [I]->[2*I+1],
                                        [I]->[2*i+2];
  };
  …
};
```

The following performance model definition represents a dynamic communication pattern,

```
algorithm DynamicPattern(int p, int pattern) {
  coord I=p;
  node {I>=0: bench*I;};
  link {
    pattern==STAR: length*(I*sizeof(double)) [0]->[I];
    pattern==RING: length*(I*sizeof(double))
                          [I]->[(I+1)/p];
   };
 };
```

which describes the star or ring communication topology depending on parameter **pattern**.

Parent Declaration

Line 14 is a *parent declaration*. It specifies the coordinates of the parent processor node in a given coordinate system. If a performance model declaration does not contain a parent node declaration, the parent is specified implicitly and has zero number in the natural numeration of processor nodes. The parent is the so-called virtual host-processor, which always maps onto the host-process associated with the user's terminal.

If we need the parent of the performance model **`pdgemm`** to have not the least but the greatest coordinates, line

```
parent [p-1,q-1];
```

has to be used in the definition of performance model **`pdgemm`** instead of line

```
parent [0,0];
```

The HMPI groups are not absolutely independent of each other. Every newly created group has exactly one process shared with already existing groups. That process is called a *parent* of this newly created group, and is the connecting link, through which results of computations are passed if the group ceases to exist.

Scheme Declaration

Line 15 introduces the *scheme declaration*. The **scheme** block describes how exactly virtual processors interact during the execution of the algorithm. The scheme block is composed mainly of two types of units. They are computation and communication units. Each computation unit is of the form $e\%\%[i]$ specifying that *`e`* percent of the total volume of computations is performed by the virtual processor with the coordinates (*`i`*). Each communication unit is of the form $e\%\%[i] \rightarrow [j]$ specifying transfer of data from virtual processor with coordinates *`i`* to the virtual processor with coordinates *`j`*. There are two types of algorithmic patterns in the scheme declaration. They are sequential and parallel. Some examples of the sequential algorithmic patterns are

```
for (e1; e2; e3) a

for (e1; e2; e3)
  for (ee1; ee2; ee3)
      a

if (e) a1 else a2
```

The parallel algorithmic patterns are specified by the keyword **par** and they describe parallel execution of some actions (mixtures of computations and communications). Some examples of the parallel algorithmic patterns are

```
par (e1; e2; e3) a

par (e1; e2; e3)
  par (ee1; ee2; ee3)
      if (e) a1 else a2
```

The **scheme** declaration describes `(n/b)` successive steps of the algorithm. At each step `k`,

- Lines 21-23 describe communications related to matrix *A*. A column of `b`×`b` blocks of matrix *A* is communicated horizontally. If processors P_{IJ} and P_{KJ} are involved in this communication so that P_{IJ} sends a part of this column to P_{KJ}, then the number of `b`×`b` blocks transferred from P_{IJ} to P_{KJ} will be `(n/(b×p))`. The total number of `b`×`b` blocks of matrix *A*, which processor P_{IJ} sends to processor P_{KJ}, is `(n/(b×p))×(n/(b×q))`.Therefore, $\frac{(n/(b\times p))}{(n/(b\times p))\times(n/(b\times q))}\times 100=\frac{1}{(n/(b\times q))}\times 100$ percent of data that should be in total sent from processor P_{IJ} to processor P_{KJ} will be sent at the step. The second nested `par` statement in the main `for` loop of the `scheme` declaration specifies this fact. Again, we use the `par` algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.
- Lines 24-27 describe communications related to matrix *B*. A row of `b`×`b` blocks of matrix *B* is communicated vertically. For each pair of abstract processors P_{IJ} and P_{IL} involved in this communication, P_{IJ} sends a part of this row to P_{IL}. The number of `b`×`b` blocks transferred from P_{IJ} to P_{IL} will be `(n/(b×q))`. The total number of `b`×`b` blocks of

matrix *B*, which processor P_{IJ} sends to processor P_{IL}, is **`(n/(b×p))×(n/(b×q))`**. Therefore, $\frac{(n/(b\times q))}{(n/(b\times p))\times(n/(b\times q))}\times 100=\frac{1}{(n/(b\times p))}\times 100$ percent of data that should be in total sent from processor P_{IJ} to processor P_{IL} will be sent at the step. The first nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies this fact. The **`par`** algorithmic patterns are used to specify that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.

- Lines 28-30 describe computations. Each abstract processor updates each its **`b`**×**`b`** block of matrix *C* with one block from the pivot column and one block from the pivot row, so that each block c_{ij} $(i, j \in \{1,\dots,n\})$ of matrix *C* will be updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. The processor performs the same volume of computation at each step of the algorithm. Therefore, at each of **`(n/b)`** steps of the algorithm the processor will perform $\frac{100\times b}{n}$ percent of the volume of computations it performs during the execution of the algorithm. The third nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies this fact. The **`par`** algorithmic patterns are used here to specify that all abstract processors perform their computations in parallel.

The **scheme** declaration shown is relatively simple. It just reflects the relative simplicity of the underlying parallel algorithm. In general, the performance model definition language allows the programmer to describe quite sophisticated heterogeneous parallel algorithms by means of wide use of parameters, locally declared variables, functions, expressions and statements.

3.1.3 Structure of Target Program

The transformed HMPI application performs typically the following steps. The HMPI calls are explained in detail in the sections that follow.

1. The initialization of HMPI runtime using the function **`HMPI_Init`**.
2. This is followed by dynamic refreshment of the estimation of the processor speeds using the function **`HMPI_Recon`**. The benchmark code used in the call to **`HMPI_Recon`** is a serial BLAS version of the parallel ScaLAPACK routine. In this case, the BLAS routine **`DGEMM`** multiplying two dense square matrices is used to dynamically refresh the processor speeds. An interesting issue is the choice of size of the matrix that is to be used in the benchmark code. An approximation of the size of the matrix used is equal to the size of the matrix used in the parallel application divided by the square root of the total number of processes that are available for computation. For example, if the total number of processes available for computation are **`m`** and the size of the matrix to be solved is **`n`**, the size of the matrix used in the benchmark code in the call to **`HMPI_Recon`** can be **`n/`**$\sqrt{m}$. This is an approximation because the optimal number of processes that can execute the parallel application (detected by **`HMPI_Group_auto_create`**) may not be equal to the total number processes available for computation.
3. Creation of a HMPI group of processes using the function **`HMPI_Group_auto_create`** to obtain a handle to the HMPI group of MPI processes. This function detects the optimal number of processes that can execute the parallel application, that is, finds the optimal arrangement of processes in a grid. During the creation of a HMPI group of processes, the mapping of the parallel processes to the executing network of computers is performed such that the number of processes running

on each processor is proportional to its speed. In other words, while distributed evenly across parallel processes, data and computations are distributed unevenly over processors of the heterogeneous network, and this way each processor performs the volume of computations proportional to its speed. The function calls **`HMPI_Is_host`**, **`HMPI_Is_free`**, and **`HMPI_Is_member`** are explained in sections on HMPI group management functions.

4. Conversion of the handle to the HMPI group of MPI processes obtained previously to an MPI communicator using the function call **`HMPI_Get_comm`**.
5. Conversion of the MPI communicator to an integer BLACS handle, which can be passed into grid creation routine. This is done using the interim BLACS routine **`Csys2blacs_handle`**.
6. Creation of the BLACS context using the integer BLACS handle. This is done using the interim BLACS routine **`Cblacs_gridinit`**.
7. The four basic steps involved in calling a ScaLAPACK routine described previously are then performed.

An interesting issue is the choice of the total number of processes to be allocated to each participating computer when the user starts up the application. Some basic rules to choose the number of processes to allocate per each processor can be followed:

1. First of all, the number of processes running on each computer should not be less than the number of processors of the computer just to be able to exploit all the available processor resources.
2. The upper bound on the number of processes to allocate per computer is limited by the underlying operating system and/or the underlying MPI implementation. For example,

Name (Number of Processors)	**Architecture**	**cpu MHz**	**Total Main Memory (mBytes)**	**Cache (kBytes)**
pg1cluster01 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
pg1cluster02 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
pg1cluster03 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
pg1cluster04 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
csserver (4)	Linux 2.6.5-1.358smp Intel(R) XEON(TM)	2867	8192	512

Table 3.1: Specifications of the five computers used for running a simple HMPI application involving HMPI group creation and destruction.

```
int main(int argc, char **argv)
{
   int p;
   HMPI_Group gid;
   HMPI_Init(argc, argv);
   if (HMPI_Is_host())
      p = 5;
   if (HMPI_Is_host())
      HMPI_Group_create(&gid, &HMPI_Model_simple, &p);
   if (HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_simple, NULL);
   if (HMPI_Is_member(&gid))
      HMPI_Group_free(&gid);
   HMPI_Finalize(0);
}
```

Figure 3.4: A simple HMPI application that calls HMPI runtime initialization, group creation and HMPI runtime finalization.

the *LAM/MPI 7.1.1* installed under the operating systems *Solaris 2.9* and *Linux 2.6.8-1.521smp* does not specify any limit on the number of LAM processes that can beexecuted on each processor. Therefore the maximum number of processes in this case is

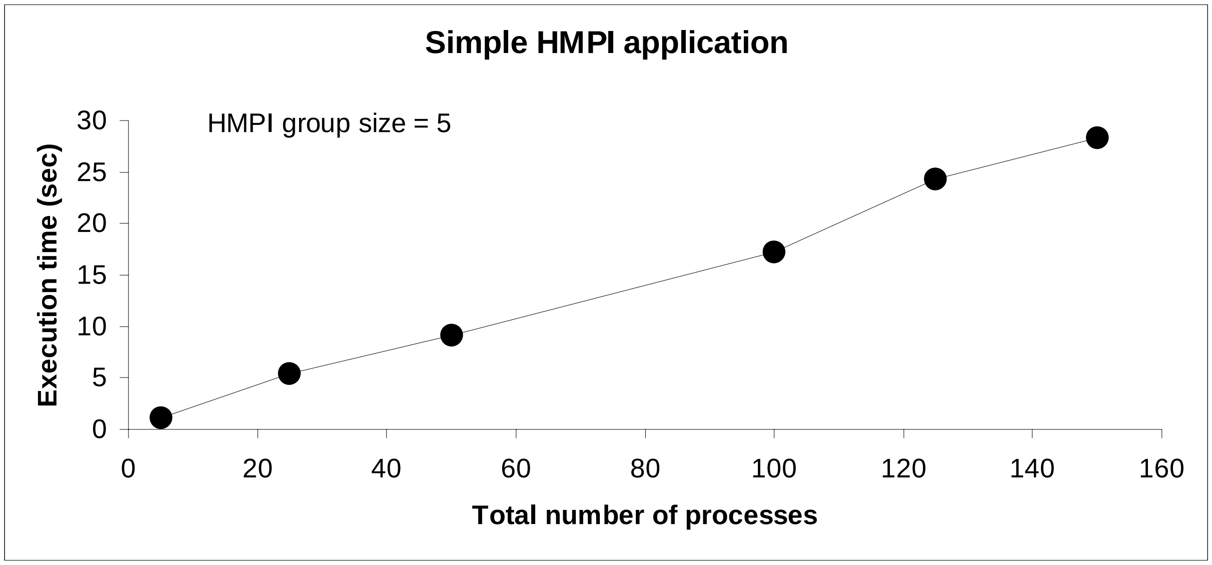


Figure 3.5: Demonstration of the influence of the growth of the number of processes on the overhead associated with HMPI group creation and destruction. The execution times are for a simple HMPI application shown in Figure 3.4 run on the network shown in Table 3.1. Only 5 processes are members of the HMPI group and are involved in the execution of the algorithm whereas the rest of the processes are idle and not involved in any computations. For the first point, a process is run on each computer of the network. For experimental point **i** (**i**>1), 5×**i** processes are run on each computer of the network.

limited by the operating system and can be obtained by using the UNIX command '`ulimit -u`'. However, the upper bound on the number of processes executed on each processor is roughly equal to the ratio of speed of the fastest processor to speed of the slowest processor on the executing network of computers.

3. The other factor affecting the execution performance of the application is the size of main memory on the computer. It must be ensured that the sum total of the amount of main memory used by all the processes allocated to a computer not exceed the size of main memory of the computer. This is because when the data size of the application is larger than the main memory size of the computer, the performance is adversely affected

Name (Number of Processors)	**Architecture**	**cpu MHz**	**Total Main Memory (mBytes)**	**Cache (kBytes)**
pg1cluster01 (1)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
pg1cluster02 (1)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
pg1cluster03 (1)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
pg1cluster04 (1)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512
csultra01 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra02 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra03 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra04 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra05 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra06 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra07 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048

Table 3.2: Specifications of the eleven computers used for running a simple HMPI application involving HMPI group creation and destruction (only one process run per processor).

```
int main(int argc, char **argv) {
   int p;
   HMPI_Group gid;
   HMPI_Init(argc, argv);
   if (HMPI_Is_host())
      p = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
   if (HMPI_Is_host())
      HMPI_Group_create(&gid, &HMPI_Model_simple, &p);
   if (HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_simple, NULL);
   if (HMPI_Is_member(&gid))
      HMPI_Group_free(&gid);
   HMPI_Finalize(0);
}
```

Figure 3.6: A simple HMPI application that calls HMPI runtime initialization, group creation and HMPI runtime finalization (HMPI group consists of one process per computer).

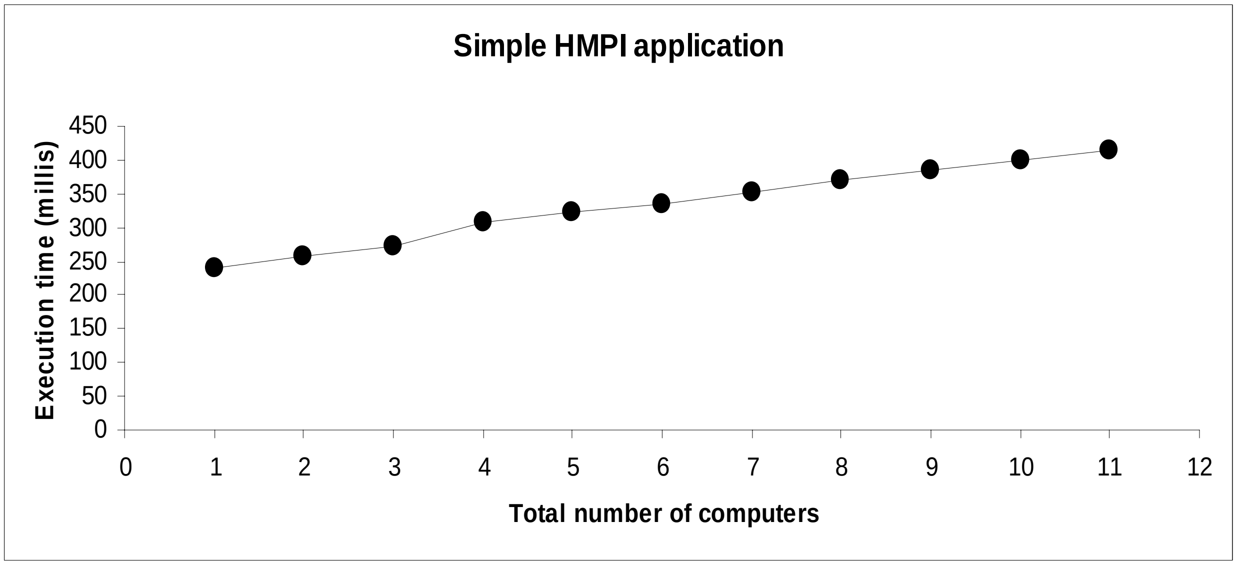


Figure 3.7: Demonstration of the influence of the growth of the number of computers on the overhead associated with HMPI group creation and destruction. The execution times are for a simple HMPI application shown in Figure 3.6 run on the network shown in Table 3.2.

because the dominant computation times were used by the operating system to do context switch and page swapping between main memory and disk.

If an HMPI application does not define a significant amount of static data, then all the processes, which are not involved in the execution of the parallel algorithm, are very light-weighted and do not consume too many resources such as processor cycles or memory. The overheads associated with these processes are the initialization and finalization of HMPI runtime and the communications associated with the synchronizations involved during the creation and the destruction of the HMPI groups. It is observed that these overheads do not grow rapidly with the growth of the total number of processes but is more sensitive to the number of computers used. Figure 3.5 shows that the growth of the number of processes does not result in a large increase in the overheads. The experiments are performed on local network of 5 Linux

Name (Number of Processors)	Architecture	cpu MHz	Total Main Memory (mBytes)	Cache (kBytes)	Absolute Speed (dgemm) (MFlops)	Absolute Speed (dgesv) (MFlops)
pg1cluster01 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512	2429	2139
pg1cluster02 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512	2429	2139
pg1cluster03 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512	2429	2139
pg1cluster04 (2)	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	2048	1024	512	2429	2139
zaphod (1)	Linux 2.4.18-3 i686 Intel Pentium III	997	256	256	563	494
maxft (1)	Linux 2.6.5-1.358 Pentium III	731	128	256	412	392

Table 3.3: Specifications of six computers of a heterogeneous network to determine the influence of blocking factor on HMPI+ScaLAPACK application.

computers shown in Table 3.1. Figure 3.7 shows the influence of the growth of the number of processes on the overhead associated with HMPI group creation and destruction. The experiments are performed on local network of eleven heterogeneous computers shown in Table 3.2.

There are two other important issues, one is the optimal arrangement of processes in the grid and the other is the blocking factor used to distribute the rows and the columns of the matrices involved in the computation. The optimal arrangement of processes in the process grid is determined by the HMPI function **`HMPI_Group_auto_create`**.

To determine the optimal blocking factor to be used to distribute the rows and the columns of the matrices involved in the computation, a heterogeneous local network of 6 Linux computers

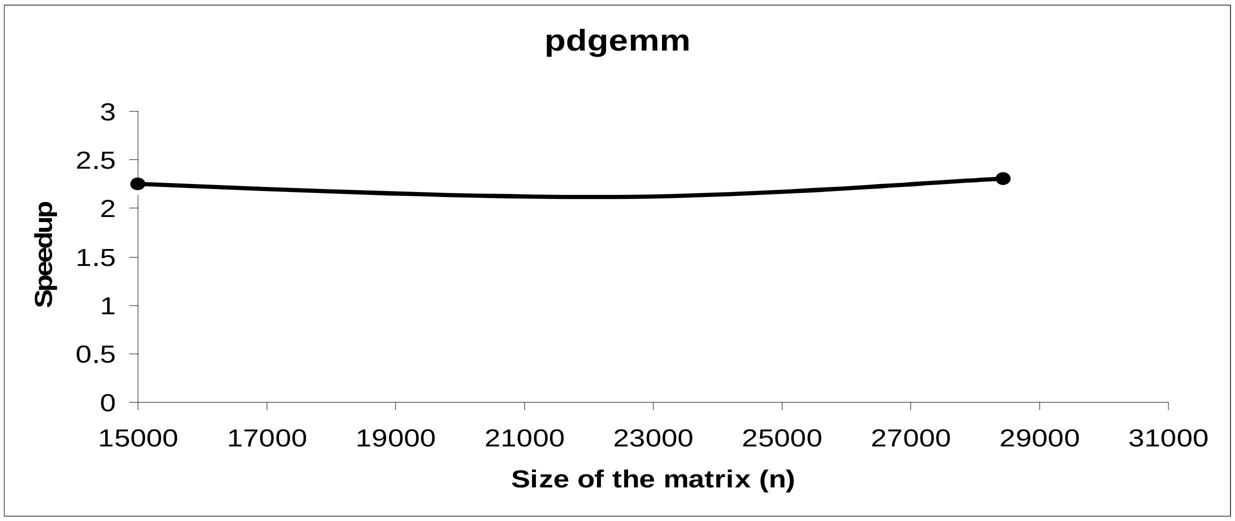
n \ **b**	**18**	**36**	**54**	**72**	**90**	**180**
1080	7.19	6.52	7.28	7.55	6.98	7.14
2160	18.39	19.35	20.18	19.83	18.94	20.05
3240	48.52	48.77	49.38	48.33	47.33	47.35
4320	101.56	100.43	100.77	104.72	100.44	104.91
5400	190.81	188.37	203.19	195.60	189.64	181.68

Table 3.4: Results of experiments on network shown in Table 3.3. **`n`** is the size of the matrix. **`b`** is the distribution blocking factor. The execution times of the parallel matrix-matrix multiplication obtained by executing the routine **`pdgemm`** are given in seconds. The process grid used in the experiments is **`p=3,q=3`** (one process per processor configuration).

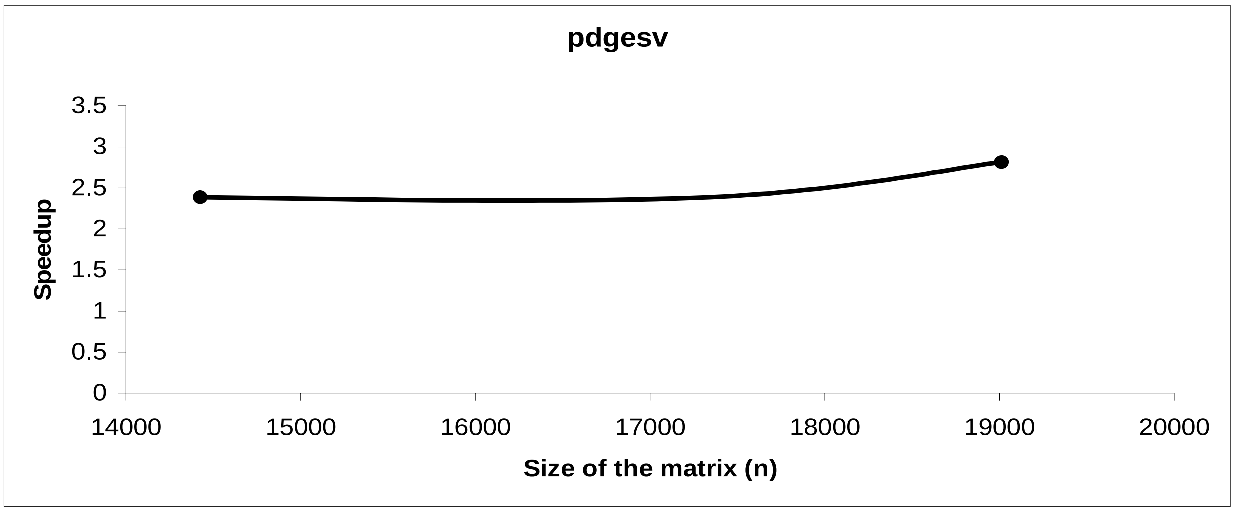
n \ **b**	**18**	**36**	**54**	**72**	**90**	**180**
1080	7.71	7.49	8.17	8.27	7.78	8.43
2160	15.30	13.62	14.47	13.21	13.10	13.31
3240	30.75	23.14	23.54	22.85	22.37	24.66
4320	55.68	48.41	48.78	45.80	44.96	45.07
5400	99.22	86.21	81.80	76.05	82.20	81.55

Table 3.5: Results of experiments on network shown in Table 3.3. **`n`** is the size of the matrix. **`b`** is the distribution blocking factor. The execution times obtained by executing the routine **`pdgesv`** are given in seconds. The process grid used in the experiments is **`p=3,q=3`** (one process per processor configuration).

shown in Table 3.3 is used in the experiments. The computers used in the experiments are connected to communication network, which is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The experimental results are obtained by averaging the execution times over a number of experiments. It is observed that the execution times are the same no matter what algorithmic blocking factor is used. Table 3.4 shows the experimental results using the routine **`pdgemm`** performing parallel matrix-matrix multiplication.



(a)



(b)

Figure 3.8: Results obtained using the network of heterogeneous computers shown in Table 3.3. (a) Comparison of speedups of matrix-matrix multiplication using the routine **`pdgemm`**. (b) Comparison of speedups of solving linear system of equations using the routine **`pdgesv`**.

Table 3.5 shows the experimental results using the routine **`pdgesv`**, which computes the solution to a real system of equations.

However to ensure efficient data distribution, it is recommended that any blocking factor between 32 to 64 be used to distribute the rows and the columns of the matrices involved in the computation of the linear algebra kernel [BCC+97].

Figure 3.8(a) shows the experimental results using the routine **`pdgemm`** performing parallel matrix-matrix multiplication on the heterogeneous network shown in Table 3.3. Figure 3.8(b) shows the experimental results using the routine **`pdgesv`**, which computes the solution to a real system of equations on the heterogeneous network shown in Table 3.3. The speedup calculated is the ratio of the execution time of the ScaLAPACK program over the execution time of the HMPI program.

The absolute speeds of the processors are obtained based on serial versions **`dgemm`** and **`dgesv`** of the corresponding parallel routines **`pdgemm`** and **`pdgesv`**. For **`dgemm`**, the size of the square matrix used is **`1000×1000`**. For **`dgesv`**, the size of the square matrix used is **`2000×2000`**. It can be seen that the fastest processors are on the *pg1cluster* computers and the slowest processor is *maxft*. The number of processes to be run on each processor is equal to the ratio of the absolute speed of the fastest processor to the absolute speed of the slowest processor. For the experiments shown in Figure 3.8, the number of processes run on each processor in *pg1cluster* is 6, on *zaphod* is 2, and on *maxft* is 1. This is because each processor on *pg1cluster* computers is 6 times faster than the processor on *maxft* and the processor on *zaphod* is 2 times faster than the processor on *maxft*. So the total number of processes available to the HMPI program for computation is 6×8 + 2×1 + 1×1 = 51.

The HMPI program detects the optimal process grid arrangement from the set of possible 2D process grid arrangements of 51 processes. Since the number of 2D process grid arrangements is large, the HMPI program uses the HMPI function **`HMPI_Group_heuristic_auto_create`** instead of the HMPI function **`HMPI_Group_auto_create`**, which evaluates all the possible 2D process grid arrangements. The function **`HMPI_Group_heuristic_auto_create`** uses heuristics to reduce the number of process arrangements to evaluate. The heuristics used are that for a 2D process grid arrangement, the row and column distribution blocking factors must be an integer multiple of the number of processes along the row of the grid and number of processes along the column of the grid respectively. The ScaLAPACK program uses a **2×5** grid of processes (using one process per processor configuration).

It should also be noted that the HMPI functions **`HMPI_Group_auto_create`** and **`HMPI_Group_heuristic_auto_create`** find the optimal process arrangement and not the optimal number of processes to run on each processor.

The Figures 3.8(a) and (b) show the speedup of the HMPI programs over ScaLAPACK programs.

3.2 HMPI Group Management Functions

Having provided such a description of the performance model, application programmers can use a new operation, whose interface is shown below, which tries to create a group that would execute the heterogeneous algorithm faster than any other group of processes,

```
HMPI_Group_create(HMPI_Group* gid,
     const HMPI_Model* perf_model,
     const void* model_parameters)
```

where **`perf_model`** is a handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model, **`model_parameters`** are the parameters of the performance model. This function returns an HMPI handle to the group of MPI processes in **`gid`**.

Users can use **`gid`**, the HMPI handle to the group of MPI processes to perform various group management operations. This handle is intended to be opaque to the application. The users should not attempt to predict its values or modify it without using functions supplied with this library.

```
algorithm Nbody(int p, int k, int n[p])
{
   coord I=p;
   node {
     I>=0: bench*((n[I]/k)*(n[I]/k));
   };
   link {
     I>0: length*(n[I]*sizeof(Body)) [I]->[0];
   };
   parent[0];
   scheme {
     int i;
     par (i = 0; i < p; i++) 100%%[i];
     par (i = 0; i < p; i++) 100%%[i]->[0];
   };
};
```

Figure 3.9: Specification of the performance model of the parallel algorithm of the simulation of evolution of bodies in the HMPI's performance definition language.

```
int main(int argc, char **argv) {
   int p, k, i, *model_params, *nbodies;
   HMPI_Group gid;
   HMPI_Init(argc, argv);
   if (HMPI_Is_host()) {
      // First parameter to the performance model
      model_params[0] = p;
      // Second parameter to the performance model
      model_params[1] = k;
      // Values of third vector parameter
      // to the performance model
      for (i = 0; i < p; i++)
         model_params[1+1+i] = nbodies[i];
   }
   if (HMPI_Is_host())
      HMPI_Group_create(&gid, &HMPI_Model_Nbody, model_params);
   if (HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_Nbody, NULL);
   if (HMPI_Is_member(&gid)) {
      // computations and communications are performed here
      …
   }
   …
   HMPI_Finalize(0);
}
```

Figure 3.10: The most principal code of the HMPI program illustrating the creation of the optimal group of processes using the operation **`HMPI_Group_create`**.

In HMPI, groups are not absolutely independent of each other. Every newly created group has exactly one process shared with already existing groups. That process is called a *parent* of this newly created group, and is the connecting link, through which results of computations are passed if the group ceases to exist. **`HMPI_Group_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HMPI group.

To illustrate the usage of the function **`HMPI_Group_create`**, consider the HMPI application shown in Figures 3.9 and 3.10. This program simulates the evolution of a system of stars in a galaxy (or a set of galaxies) under the influence of Newtonian gravitational attraction.

Consider the block of code containing the call to **`HMPI_Group_create`** shown in Figure 3.10. The first parameter **`gid`** is an output parameter, which is an HMPI handle to the group of MPI processes. The second parameter **`HMPI_Model_Nbody`** is an input parameter, which is a handle to the performance model. It is a structure generated by the performance model definition language compiler from the compilation of the performance model definition shown in Figure 3.9. The generated code is shown in appendix A. In the current implementation of HMPI, the scalar and the vector parameters to the performance model must be of type integer. Vector parameters can be multidimensional arrays. However, the declaration of any vector parameter with dimensions parameterized by scalar parameters must follow the declaration of the scalar parameters. In the example shown in Figure 3.9, the vector parameter **`n`**, which is an indexed set of integers of size **`p`**, follows the scalar parameter **`p`**.

The third parameter **`model_params`** is a one-dimensional array containing all the values of the parameters to the performance model. As can be seen from the Figure 3.10, the host process fills the parameter **`model_params`** with the first parameter to the performance model, which is the number of processes **`p`** involved in the execution of the heterogeneous algorithm, the second parameter to the performance model, which is the number of bodies in the group used in the benchmark code, and finally the last parameter to the performance model, which is the vector parameter of **`p`** groups where **`i`**-th element contains the number of bodies in **`i`**-th group. Only the parent of the group, which is the host process in this case, need only fill in the model parameters.

`HMPI_Group_create` is a collective operation called by the parent and all the processes, which are not members of any HMPI group. The parent of the group in this case is the host process. The host process is a member of the pre-defined HMPI group **`HMPI_HOST_GROUP`**. This group consists of exactly one virtual processor, which always maps onto the host process associated with the user's terminal. The function **`HMPI_Is_host`** returns **`true`** if the process calling this function is the host process otherwise **`false`**. The function **`HMPI_Is_free`** returns **`true`** if the process is free, that is, the process is not a member of any group and **`false`** otherwise.

The application programmers should avoid using groups created with the MPI group constructor operations, to perform computations and communications in parallel with HMPI groups, as it may not result in the best execution performance of the application. The point is that the HMPI runtime system is not aware of any group of MPI processes, which is not created under its control. Therefore, the HMPI runtime system cannot guarantee that an HMPI group will execute its parallel algorithm faster than any other group of MPI processes if some groups of MPI processes, other than HMPI groups, are active during the algorithm execution.

Application programmers must use the group destructor operation, whose interface is shown below, to free resources associated with a group,

```
HMPI_Group_free(HMPI_Group* gid)
```

where **`gid`** is the HMPI handle to the group of MPI processes. This is a collective operation and must be called by all the members of this group. There are no analogs of other group constructors of MPI such as the set-like operations on groups and the range operations on groups in HMPI. This is because:

- Firstly, HMPI does not guarantee that groups composed using these operations can execute a logical unit of parallel algorithm faster than any other group of processes, and
- Secondly, it is relatively straightforward for application programmers to perform such group operations by obtaining the groups associated with the MPI communicator given by the **HMPI_Get_comm** operation, whose interface is shown below.

  ```
  const MPI_Comm* HMPI_Get_comm (
       const HMPI_Group* gid)
  ```

  This function returns an MPI communicator with communication group of MPI processes defined by **`gid`**. This is a local operation not requiring inter-process communication.

The other additional group management operations provided by HMPI apart from the group constructor and destructor are the following group accessors:

- **`HMPI_Group_rank`** to get the rank of the process in the HMPI group,
- **`HMPI_Group_size`** to get the number of processes in this group,
- **`HMPI_Group_parent`** to get the rank of the parent of this group, and
- **`HMPI_Group_performances`** to get the relative speeds of the processes in this group.

When describing the features of the performance model definition language, we mentioned that the declaration of the performance model is also called a *topology*. Thus in HMPI, there is an implicit virtual process topology associated with HMPI groups. The operations used to get the information about the topology of the HMPI group of processes are:

- **`HMPI_Group_topo_size`** to get the number of dimensions of the process arrangement of the virtual process topology of the HMPI group,
- **`HMPI_Group_topology`** to get the number of processes in each dimension of the process arrangement of the virtual process topology of this HMPI group,
- **`HMPI_Group_coordof`** to get the coordinates of the process in the virtual process topology of this HMPI group,
- **`HMPI_Rank`** to get the rank of the process in a group given its coordinates in the virtual process topology of this HMPI group, and
- **`HMPI_Coordof`** to get the coordinates of the process in the virtual process topology of the HMPI group given the rank of the process in this HMPI group.

3.2.1 Mapping Algorithm

During the creation of this group of processes, HMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network. The solution to the problem is based on the following:

- The performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model.

- The performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm. This model considers the executing heterogeneous network as a multilevel hierarchy of interconnected sets of heterogeneous multiprocessors. This model takes into account the material nature of communication links and their heterogeneity.

3.2.1.1 Model of a Heterogeneous Network of Computers

The model of a heterogeneous network of computers allows for the material nature of communication links and their heterogeneity. Each computer in this model is characterized by two attributes:

- The time of execution of a (serial) test code on the computer;
- The number of physical processors.

The first attribute is a function of time, and it can vary even during the execution of the same HMPI application. The second attribute is a constant and it determines how many noninteracting processes can run in parallel on the computer without loss of speed.

The model considers the executing heterogeneous network as a multilevel hierarchy of interconnected sets of heterogeneous multiprocessors. The hierarchy reflects the heterogeneity of communication links and can be represented in the form of an attributed tree.

Each node of the tree represents a homogeneous communication space of the heterogeneous network. The first attribute associated with an internal node is the set of computers, which is just a union of sets of computers associated with its children.

The second is the speed of data transfer between two computers from different sets associated with its children. This attribute characterizes point-to-point communication at this communication layer and is a function of size of the transferred data block, **s(d)**. Note, that **s(0)** is not zero and equal to startup time of point-to-point communication at this layer.

The third attribute specifies if the communication layer allows parallel point-to-point communications between different pairs of computers without loss of data transfer speed, or the layer serializes all communications. This attribute can have two values – **Serial** and **Parallel**. A pure Ethernet network is serial. At the same time, the use of switches can make it parallel.

The next group of attributes is only applicable to a parallel communication layer. It characterizes collective communication operations such as broadcast, scatter, gather, and reduction. The point is that a collective communication operation cannot be considered as a set of independent point-to-point communications. It normally has some special process, called *root*, which is involved in more communications than other participating processes.

The level of parallelism of each of the collective communication operations depends on its implementation and is reflected in the model by means of the corresponding attribute. For example, the attribute $\mathbf{f_b}$ characterizes the level of parallelism of the broadcast operation. It is supposed that the execution time **t** of this operation can be calculated as follows

$$t = f_b * t_p + (1- f_b)*t_s$$

where t_s is the time of purely serial execution of the operation, and t_p is the time of ideally parallel execution of this operation ($0 <= f_b <= 1$).

Each leaf node of this tree represents a single (homogeneous) multiprocessor computer.

This communication model addresses **n**-level heterogeneous parallel systems, where **n** represents the number of network layers present in the heterogeneous environment. This model describes multiple heterogeneous parallel computers connected by some combination of internal buses, local-area networks, campus-area networks, and wide-area networks. As a result, it can guide the design of applications for traditional parallel systems, heterogeneous or homogeneous

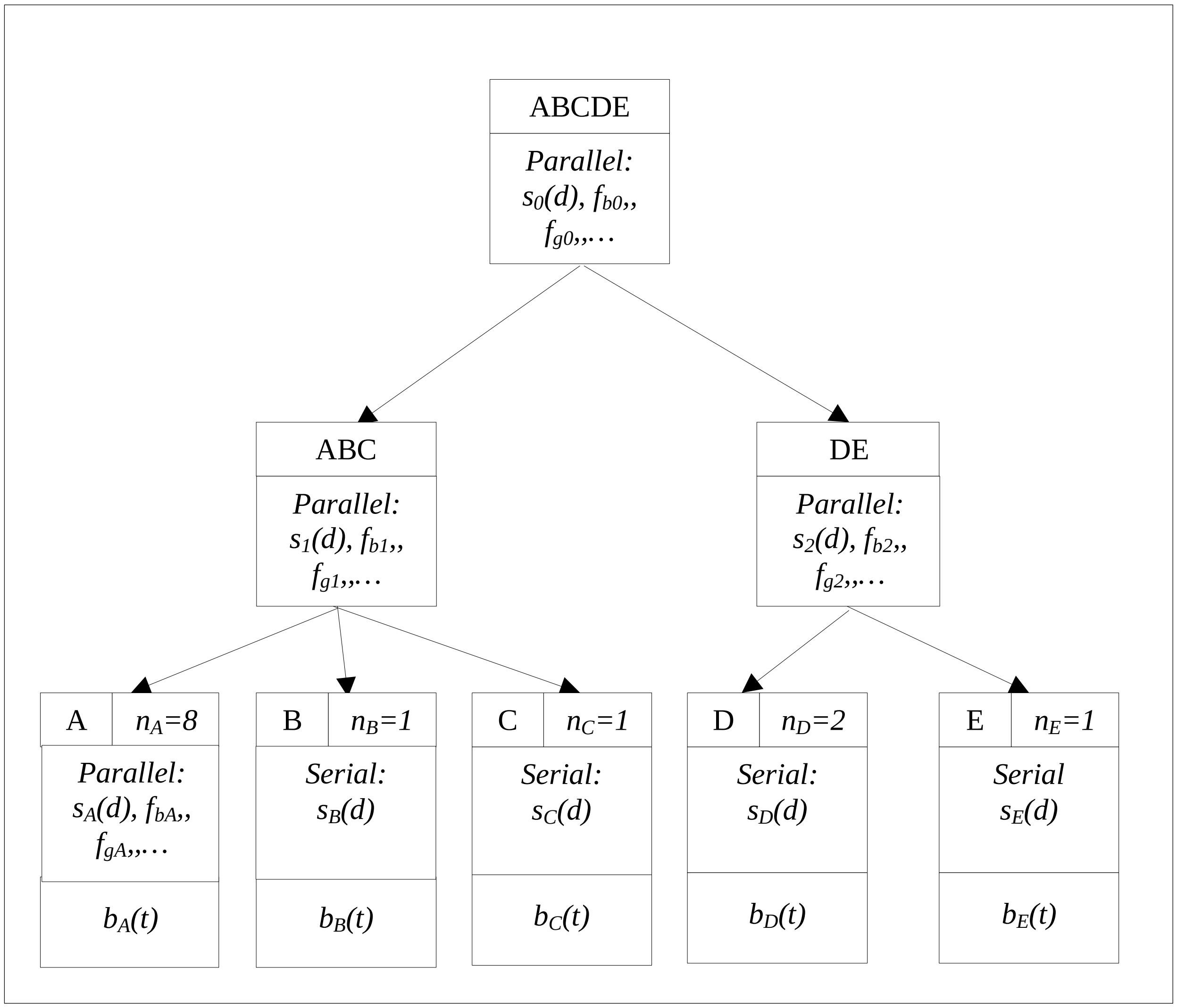

ABCDE
Parallel: $s_0(d)$, f_{b0},, f_{g0},,…
ABC
Parallel: $s_1(d)$, f_{b1},, f_{g1},,…
DE
Parallel: $s_2(d)$, f_{b2},, f_{g2},,…
A $n_A=8$
Parallel: $s_A(d)$, f_{bA},, f_{gA},,…
$b_A(t)$
B $n_B=1$
Serial: $s_B(d)$
$b_B(t)$
C $n_C=1$
Serial: $s_C(d)$
$b_C(t)$
D $n_D=2$
Serial: $s_D(d)$
$b_D(t)$
E $n_E=1$
Serial $s_E(d)$
$b_E(t)$

Figure 3.11: Hierarchical model of a heterogeneous network of five computers.

clusters, the Internet, and computational grids. Furthermore, this model incorporates parameters that reflect the relative computational and communication speeds at each of the **n** levels.

Figure 3.11 depicts the model for a local network of 5 computers, named A, B, C, D and E. Computer A is a distributed-memory 8-processor computer, D is a shared-memory 2-processor server. Computers B, C and E are uniprocessor workstations. The local network consists of 2 segments with A, B and C belonging to the first segment. Computers D and E belong to the second segment.

The speed of transfer of a data block of k bytes from a process running on computer C to a process running on computer D is estimated by $s_0(k)$, meanwhile the speed of transfer of the same data block from a process running on computer C to a process running on computer A is estimated by $s_1(k)$.

The level of parallelism of a broadcast involving processes running on computers B, C, and E is f_{b0}, meanwhile that of a broadcast involving processes running on computer A is f_{bA}.

The communication model presented is simple and rough enough. It is used at runtime by the HMPI programming system to predict the execution time of the implemented parallel algorithm. It uses a small number of integral attributes presenting some average characteristics rather than detailed and fine-structured description.

The main reason of this simplicity is that the target architecture for HMPI is common networks of computers, which normally are multi-user environments of irregular structure with not very stable characteristics. Therefore, fine-grained communication effects can hardly be reliably predicted for that architecture.

Secondly, HMPI is aimed at programming applications, in which computations prevail over communications, i.e., the contribution of computations in the total execution time is much higher than that of communications. If it is not the case, it normally means that the main goal of the application is not to speed up the solution of some individual problem, and the distribution of its components over different computers is its intrinsic feature, i.e., the application is actually distributed not parallel one.

Thus, HMPI needs an efficient communication model of common heterogeneous networks of computers suitable for prediction of the execution time of data transfer operations involving the transfer of relatively big volumes of data. The accuracy of the prediction does not need to be too high because the contribution of communications in the total execution time is supposed to be relatively small. Actually, the accuracy cannot be high because of the nature of the modelled hardware.

The main disadvantage of the communication model that should be addressed in the future work is that it is static. An efficient way to update its parameters at runtime to reflect the current situation could improve its accuracy. Another possible direction of improvement is the model of parallel communication layer and collective communication operations. More experiments with different network configurations are needed to make the model more accurate for a wide range of common networks.

3.2.1.2 Overview of the Mapping Algorithm

The algorithms used to solve the problem of selection of processes are discussed in [Las02]. We describe the mapping algorithm here in order to make this composition self-contained.

Each particular mapping, $\mu{:}I\text{->}C$, where I is a set of processes of the group, and $C=\{c_0, c_1,\ldots,c_{\mathbf{M}-1}\}$ is a set of computers of the executing network, is characterized by the estimation of the time of execution of the algorithm on the network of computers. The estimation is calculated based on the performance model of the parallel algorithm and the model of the executing network of computers.

Ideally, the HMPI runtime system should find such a mapping that is estimated to ensure the fastest execution of the parallel algorithm. In general, for an accurate solution of this problem as many as $\boldsymbol{M}^{\boldsymbol{K}}$ possible mappings have to be probated to find the best one (here, $\boldsymbol{K}$ is the power of the set ***I*** of processes of the group). Obviously, that computational complexity is not acceptable for a practical algorithm that should be performed at runtime. Therefore, the HMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time, namely, after probation of $\boldsymbol{M}\times\boldsymbol{K}$ possible mappings instead of $\boldsymbol{M}^{\boldsymbol{K}}$.

The underlying algorithm is the following. At the preliminary step, the set I is re-ordered in accordance with the volume of computations to be performed by the virtual processors, so that the most loaded virtual processor will come first. Let $P=\{p_k\}(k=0,\ldots,K-1)$ be this well-ordered set. Let Q_j be a subgroup of the abstract group formed by the set $P_j=\{p_i\}(i=0,\ldots,j)$ of virtual processors. By definition, a subgroup is a result of projection of the abstract group onto some subset of its virtual processors. Semantically, the subgroup is equivalent to its supergroup modified in the following way:

- The zero volume of computations is set for each virtual processor not included in the subgroup;
- The zero volume of communications is set for each pair of virtual processors such that at least one of which not included in the subgroup.

Finally, let c_j denote the j-th computer from the set C. Then the main loop of the algorithm can be described by the following pseudo-code:

```
for(k=0; k<K; k++) {
  for(j=0, $t_{best}$=MAXTIME, $c_{best}$=$c_0$; j<M; j++) {
    if($p_k$ is not a parent of the group) {
      Map $p_k$ to $c_j$
      Estimate execution time t for this mapping of $Q_k$ to C
      if(t<$t_{best}$) {
         $t_{best}$=t;
         $c_{best}$=$c_j$;
      }
      Unmap $p_k$
    }
  }
```

```
    Map $p_k$ to $c_{best}$
  }
```

The presented algorithm reflects the focus of HMPI on applications with computations prevailing over communications. Therefore, the algorithm is driven by virtual processors not communication links. Another argument for that approach is that the maximal number of virtual communication links is equal to the total number of virtual processors squared. Therefore, in general, an algorithm driven by virtual links would be more expensive.

Informally, the algorithm first maps the most loaded virtual processor not taking into account other virtual processors as well as communications. Then, given the first virtual processor mapped, it maps the second most loaded virtual processor only taking into account communications between these two processors and so on. At the **i**-th step, it maps the **i**-th most loaded virtual processor only taking into account data transfer between these **i** virtual processors. This algorithm exploits the obvious observation that the smaller are things, the easier they can be evenly distributed. Hence, bigger things should be distributed under weaker constraints than smaller ones. For example, if you want to distribute a number of balls of different size over a number of baskets of different size, you better start from the biggest ball and put it into the biggest basket; then put the second biggest ball into the basket having the biggest free space and so on. This algorithm keeps balance between ball sizes and free basket space and guarantees that if at some step you don’t have enough space for the next ball, it simply means that there is no way to put all the balls in the baskets. Similarly, if the above algorithm cannot balance the load of actual processors in case of practically zero communication costs, it simply means that there is no way to balance them at all. This algorithm will also work well if data transfer between more loaded virtual processors is more significant than data transfer between less loaded ones. In this case, more loaded virtual communication links are taken into account at earlier stages of the algorithm.

An obvious case when this mapping algorithm may not work well is when the least loaded virtual processor is involved in transfer of much bigger volume of data than more loaded ones, and the contribution of communications in the total execution time is significant. But even quick analysis shows that it is not the case for most parallel algorithms.

3.3 Dynamic Updating of Processor Performances Using HMPI_Recon

During the creation of a group of processes, HMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network. The solution to the problem is based on the following:

- The performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model.
- The performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

The accuracy of the performance model of the executing network of computers depends upon the accuracy of the estimation of the actual speeds of processors. HMPI provides an operation to dynamically update the estimation of processor speeds at runtime. It is especially important if computers, executing the target program, are used for other computations as well. In this case, the actual speeds of processors can dynamically change dependent on the external computations. The use of this operation, whose interface is shown below, allows the application programmers to write parallel programs, sensitive to such dynamic variation of the workload of the underlying computer system,

```
typedef void (*HMPI_Benchmark_function)(
              const void*, int, void*);
HMPI_Recon(HMPI_Benchmark_function func,
       const void* input_p,
       int  num_of_parameters, void* output_p)
```

where all the processors execute the benchmark function **func** in parallel, and the time elapsed by each of the processors to execute the code is used to refresh the estimation of its speed. This is a collective operation and must be called by all the processors in the group associated with the predefined communication universe **HMPI_PROC_WORLD** of HMPI.

This interface is designed based on the **recon** statement provided by the mpC language to perform refreshment of the relative performances of processors of the executing network of heterogeneous computers. **HMPI_Recon** call is executed by all the processors, which are members of **HMPI_PROC_WORLD_GROUP**.

HMPI provides a predefined communication universe **HMPI_COMM_WORLD**, which is a communicator consisting of all processes available for the computation in a HMPI application; this communicator has the same value in all processes. It is an analog of **MPI_COMM_WORLD**, the predefined communication universe defined in the standard MPI specification. It cannot be deallocated during the life of the process. The group corresponding to this communicator is a pre-defined constant **HMPI_COMM_WORLD_GROUP**. HMPI also provides a communication universe **HMPI_PROC_WORLD**, which is a communicator consisting of one process per processor. The group corresponding to this communicator is a pre-defined constant **HMPI_PROC_WORLD_GROUP**.

A typical parallel application is composed of one or more phases, which are sections of code comprised of computations and communications. If the phases are distinct, the application programmer has to optimally distribute computations involved in each phase amongst processors involved in executing the phase. To achieve load balance in each phase, we distribute

```
void Phase1_benchmark_code(const void*, int, void*);
void Phase2_benchmark_code(const void*, int, void*);
int main() {
   …
   for (i = 0; i < number_of_iterations; i++) {
      double *phase1_speeds, *phase2_speeds;
      //Phase1
      if ((HMPI_Is_member(HMPI_PROC_WORLD_GROUP)) {
         HMPI_Recon(&Phase1_benchmark_code,…);
         HMPI_Get_processors_info(phase1_speeds);
      }
      //Distribute computations using the speeds
      HMPI_Group_create(…);
      //Execution of the computations and communications
      //Free the group
      //Phase2
      if ((HMPI_Is_member(HMPI_PROC_WORLD_GROUP)) {
         HMPI_Recon(&Phase2_benchmark_code,…);
         HMPI_Get_processors_info(phase2_speeds);
      }
      //Distribute computations using the speeds
      HMPI_Group_create(…);
      //Execution of the computations and communications
      //Free the group
      …
    }
 }
```

Figure 3.12: An example illustrating the usage of the operation **`HMPI_Recon`** to write parallel programs sensitive to dynamic changing loads.

computations in that phase such that the volume of computations that is executed by each processor is proportional to its speed. Thus if the phases are distinct, the application programmer has to determine the speeds of the processors for each phase. To determine the speeds of the processors for each phase, the application programmer provides benchmark code that is representative of the basic computations performed in that phase to **`HMPI_Recon`**. For each phase, all the processes, which are the members of **`HMPI_PROC_WORLD_GROUP`**, execute the benchmark code. The relative speeds are determined from the time taken to execute the benchmark code. It is to be noted that the computation portions of each communication operation in a phase (preparation of the message by adding header, trailer, and error correction information, execution of the routing algorithm) must be taken into account. If the phases are not distinct, it is sufficient to use the speeds determined for one phase to be used in all the phases. However if the processors used in the execution of the parallel application are used for external computations, it is recommended that **`HMPI_Recon`** be used to determine the speeds of the processors for each phase so that computations are distributed over the processors in accordance to their actual performances at the moment of execution of the computations.

The accuracy of **`HMPI_Recon`** depends upon how accurately the benchmark code provided by the user reflects the core computations of each phase. If the benchmark code provided is an accurate measurement of the core computations in each phase, **`HMPI_Recon`** gives an accurate measure of the speeds.

Figure 3.12 illustrates the usage of **`HMPI_Recon`** to write parallel programs sensitive to the dynamic variation of the workload of the underlying computer system. As can be seen from the figure, the combination of calls **`HMPI_Recon`** and **`HMPI_Group_create`** can be used for each distinct phase of the parallel application to create a group of processes that executes the computations and communications in that phase with best execution performance. One of the approaches to tackle applications that do not have very uniform iterations is to break down the non-uniform iterations in the application into sets of uniform iterations. For each such set of uniform iterations, a performance model is designed. A group of processes is then created that can execute the set of uniform iterations with best execution performance and destroyed at the end of execution of the set.

3.4 Estimation of Execution Time of an Algorithm using HMPI_Timeof

Another principal operation provided by HMPI allows application programmers to predict the total time of execution of the algorithm on the underlying hardware without its real execution. Its interface is shown below,

```
HMPI_Timeof(const HMPI_Model* perf_model,
            const void* model_parameters)
```

This function allows the application programmers to write such a parallel application that can follow different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance. This is a local operation that can be called by any process, which is a member of the group associated with the predefined communication universe **`HMPI_COMM_WORLD`** of HMPI.

This interface is designed based on the **`timeof`** operator provided by the mpC language, which predicts the total time of the algorithm execution on the underlying hardware without its real execution.

This function invokes the HMPI runtime system, which selects the optimal set of processes based on the performance model of the parallel algorithm **`perf_model`**, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm. The estimated execution time of the algorithm by this optimal set of processes is then returned. The parameters **`model_parameters`** to the performance model are usually the following:

- Number of processes in each dimension of the process arrangement.
- Data distribution parameters specifying how the data is distributed amongst the processes, and the amount of data that is transferred between the pair of processes.

The process calling this function provides this information to the HMPI runtime system, which uses it along with the model of the executing network of computers to estimate the time of execution of the algorithm.

The estimation procedure is explained in detail in [Las02] and is presented here in order to make this composition self-contained. The time of execution for each mapping, μ:*I*->*C*, where *I* is a set of the processes of the group, and $C = \{c_0, c_1, \cdots, c_{M-1}\}$ is a set of computers of the executing network, is estimated. The estimation time for the optimal mapping, which would ensure the fastest execution of the parallel algorithm, is returned. In general, for accurate solution of this problem as many as $\boldsymbol{M^K}$ possible mappings have to be probated to find the best one (here, ***K*** is the power of the set ***I*** of processes of the group). Obviously, that computational complexity is not acceptable for a practical algorithm that should be performed at runtime. Therefore, the HMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time, namely, after probation of $\boldsymbol{M \times K}$ possible mappings instead of $\boldsymbol{M^K}$.

The estimation procedure is summarized here. Each computation unit in the **scheme** declaration of the form $e\%\%[i]$ is estimated as follows:

$$\textbf{timeof}(e\%\%[i]) = (e/100) \times v_i \times b_{\mu(i)}(t_0),$$

where v_i is the total volume of computations to be performed by the virtual processor with the coordinates i, and $b_{\mu(i)}(t_0)$ is the time of execution of the benchmark code on the computer $\mu(i)$ provided by the execution of the **HMPI_Recon** call (t_0 denotes that time when this execution took place).

Each communication unit of the form $e\%\%[i] \to [j]$ specifying transfer of data from virtual processor with coordinates *i* to the virtual processor with coordinates *j* is estimated as follows:

$$\textbf{timeof}(e\%\%[i]\text{->}[j]) = (e/100) \times w_{i\text{->}j} \times s_{\mu(i)\text{->}\mu(j)}(w_{i\text{->}j}),$$

where $w_{i\text{->}j}$ is the total volume of data to be transferred from the virtual processor with the coordinates i to the virtual processor with the coordinates j, and $s_{\mu(i)\text{->}\mu(j)}(w_{i\text{->}j})$ is the speed of transfer of data block of $w_{i\text{->}j}$ bytes between computers $\mu(i)$ and $\mu(j)$.

Simple calculation rules are used to estimate the sequential algorithmic patterns in the **scheme** declaration. For example, the estimation of the pattern

```
for (e1; e2; e3) a
```

is calculated as follows:

```
for (T=0, e1; e2; e3)
   T += time taken to execute action a
```

The rules just reflect semantics of the corresponding serial algorithmic patterns. The rule to estimate time for a parallel algorithmic pattern

```
par (e1; e2; e3) a
```

is more complicated.

Let $A=\{a_0, a_1, \ldots, a_{N-1}\}$ be a set of the actions ordered in accordance with the estimation of the time of their execution, namely, **timeof**(a_0)>=**timeof**(a_1)>=…>=**timeof**(a_{N-1}). Let B be a subset of A consisting of all actions that only perform communications, $B=\{b_0, b_1, \ldots, b_{Q-1}\}$. Let $C=\{c_0, c_1, \ldots, c_{M-1}\}$. Finally, let v_i be the number of virtual processors mapped on the computer c_i, and f_i be the total number of physical processors of the computer. Then the rule to calculate the estimation **T** of the pattern looks as follows:

```
for(j=0, T=0; j<M; j++) {
  for(i=0, T0=0, k=0; k<Upper(vj, fj) && i<N; i++) {
    if(ai performs some computations on cj) {
      T0 += timeof(ai);
      k++;
    }
  }
  T = max(T, T0);
```

```
        }
        T = max(T, timeof(B));
```

Here, the function **`Upper`** is defined as follows:

```
      Upper(x, y) = if(x/y <= 1)
                    then 1
                    else if ((x/y)*y == x)
                         then x/y
                         else x/y+1
```

Informally, the above system of loops first computes for each computer the estimation T_0 of the time of parallel execution of those actions, which use that computer for some computations. The estimation is calculated, proceeding from the assumption, that if the number of parallel actions on one computer exceeds the number of its physical processors, then

- The actions are distributed evenly over the physical processors, that is, the number of actions executed by different physical processors differs by at most one;
- The most computationally intensive actions are executed on the same physical processor.

Then those parallel actions, which are not related to computations, that is, perform pure communications, are taken into account. These communication actions make up the set *B*. Let *l(B)* be the least communication layer covering all communication links involved in *B*, and let f_b, f_g be the level of parallelism of broadcast and gather correspondingly for this layer. Then the rule to calculate the estimation **T** of parallel execution of communication operations from set *B* looks as follows:

```
        if(l(B) is serial)
          for(i=0, T=0; i<Q; i++)
            T += timeof($b_i$);
        else if(B matches broadcast/scatter) {
          for(i=0, $T_{serial}$=0, $T_{parallel}$=0; i<Q; i++) {
            $T_{serial}$ += timeof($b_i$);
            $T_{parallel}$ = max($T_2$, timeof($b_i$));
          }
          T = $f_b$*$T_{parallel}$ +(1-$f_b$)*$T_{serial}$
```

```
    }
    else if(B matches gather) {
      for(i=0, $T_{serial}$=0, $T_{parallel}$=0; i<Q; i++) {
        $T_{serial}$ += timeof($b_i$);
        $T_{parallel}$ = max($T_2$, timeof($b_i$));
      }
      T = $f_g$*$T_{parallel}$ +(1-$f_g$)*$T_{serial}$
    }
    else
      for(i=0, T=0; i<Q; i++)
        T += max(T, timeof(bi));
```

The rule just sums the execution time of parallel communication operations if the underlying communication layer serializes all data packages. Otherwise we have a parallel communication layer, and if the set *B* of communication operations looks like broadcasting or scattering, i.e., one virtual processor sends data to other involved virtual processors, then the time of parallel execution of the communication operations is calculated as if they performed broadcast. Similarly, if *B* looks like gathering, i.e., one virtual processor receives data from other involved virtual processors, then the time of parallel execution of the communication operations is calculated as if they performed gather. In all other cases, it is assumed that *B* is a set of independent point-to-point communications. It is responsibility of the programmer not to specify different communication operations sharing the same communication link as parallel ones.

The rule for estimation of the execution time of the parallel algorithmic pattern is the core of the entire mapping algorithm determining its accuracy and efficiency. It takes into account material nature and heterogeneity of both processors and network equipment. It relies on fairly allocating processes to processors in a shared-memory multiprocessor normally implemented by operating systems for processes of the same priority (HMPI processes are just the case). At the same time, it proceeds from the pessimistic point of view when estimating workload of different processors of that multiprocessor. Estimation of communication cost by the rule is sensitive to scalability of the underlying network technology. It treats differently communication layers serializing data packages and supporting their parallel transfer. The most typical and widely used collective communication operations are treated specifically to provide better accuracy of the estimation of their execution time. An important advantage of the rule is its relative simplicity and effectiveness. The effectiveness is critical because the algorithm is supposed to be multiply executed at runtime.

Most disadvantages of the rule are just the backside of its simplicity and the necessity to keep it effective. Except some common collective communication operations, it is not sensitive to different collective communication patterns such as ring data shifting, tree reduction, etc., treating all them as a set of independent point-to-point communications. The main problem is that recognition of such patterns is very expensive. A possible solution is introduction in the performance model definition language some explicit constructs for communication pattern specification as a part of the scheme description. Another disadvantage of the rule affecting the accuracy of estimation is that any set of parallel communications is treated as if they all take place at the same communication layer in the hierarchy, namely, at the lowest communication layer covering all involved processors. In reality, some of the communications may use different communication layers. Incorporation of multi-layer parallel communications in this algorithm without significant loss of its efficiency is a very difficult problem, which is supposed to be addressed in future work.

`HMPI_Timeof` can thus be used to estimate the execution time on HNOCs for each possible set of model parameters **`model_parameters`**. Application programmers can use this function creatively to design best possible heuristics for the set of parameters. Depending on the time estimated for each set, the optimal values of the parameters are determined. These values are then passed to the performance model during the actual creation of the group of processes using the function **`HMPI_Group_create`**.

The accuracy of the estimation by **`HMPI_Timeof`** is dependent on the following:

- The accuracy of the performance model of the algorithm designed by the user,
- The quality of the heuristics designed for the set of parameters provided to the performance model,
- The accuracy of the performance model of the executing network of computers. This depends on the accuracy of the measurements of the processor speeds given by **`HMPI_Recon`** and the communication model of the executing network of computers. Currently the communication model used in HMPI runtime system is static. Future works would address the issue of efficiently updating the parameters of communication model at runtime.

3.5 Detection of Optimal Number of Processes using HMPI_Group_auto_create

One of the most important parameters, which influence the performance of the parallel application on HNOCs, is the number of processes used to execute the parallel application. Another principal operation provided by HMPI allows application programmers not to bother about finding the optimal number of processes that can execute the parallel application. They can specify only the rest of the parameters thus leaving the detection of the optimal number of processes to the HMPI runtime system. Its interface is shown below.

```
HMPI_Group_auto_create (
     HMPI_Group* gid, const HMPI_Model* perf_model,
     const void* model_parameters)
```

This function returns an HMPI handle to the group of MPI processes in **`gid`**.

The parameter **`perf_model`** is a handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model.

The parameter **`model_parameters`** is an input parameter. User fills the parameter **`model_parameters`** with values of the input parameters and ignores the return parameters specifying the number of processes to be involved in executing the algorithm and their relative performances.

`HMPI_Group_auto_create` is a collective operation and must be called by the parent and all the processes, which are not members of any HMPI group.

There are no restrictions imposed by the function **`HMPI_Group_create`**. It just uses the input parameters provided to create a group of MPI processes. However, the function **`HMPI_Group_auto_create`** imposes certain restrictions, which are explained below:

1. The application programmers describe a performance model of their implemented heterogeneous algorithm. The output parameters to the performance model are placed last in the list of parameters to the performance model. The output parameters are the number of processes in each dimension of the processor arrangement and an array representing the relative performances of the processors. Consider for example the performance model of an application multiplying two dense **n×n** matrices on one-dimensional processor arrangement.

```
algorithm AxB_1d(int n, int p, int speeds[p]) {
  …
};
```

In this performance model, the scalar parameter **n** is an input parameter whereas the scalar parameter **p** representing the number of processes in the linear array and the vector parameter **speeds** of size **p** representing the relative performances are the output parameters. These are the output parameters because these are determined by the function call **HMPI_Group_auto_create**. Consider for example the performance model of an application multiplying two dense **n×n** matrices on two dimensional processor grid.

```
 algorithm AxB_2d(int n, int p, int q, int speeds[p×q]) {
   …
};
```

In this performance model, the scalar parameter **n** is an input parameter whereas the scalar parameter **p** representing the number of processes in the column dimension of the processor grid arrangement, scalar parameter **q** representing the number of processes in the row dimension of the processor grid arrangement and the vector parameter **speeds** of size **p×q** representing the relative performances are the output parameters.

So generally speaking, the output parameter list contains scalar parameters, each parameter representing the number of processes in a dimension of the processor arrangement and a vector parameter representing the relative performances of the processors, the size of the vector being equal to the product of these scalar parameters.

2. If the value of any parameter used in the body of the performance model declaration is dependent on the output parameters, then it should be obtained using functions. This is mainly the case for data distribution parameters, whose values are parameterized by number of processes in each dimension of the processor arrangement and the relative performances of the processors. This is because the function **`HMPI_Group_auto_create`** executes the performance model for different processor arrangements and hence the values of the parameters dependent on the arrangement of processors and their speeds should be obtained by using functions. Consider for example the performance model of an application implementing a parallel algorithm of the simulation of evolution of **`n`** bodies on one-dimensional processor arrangement.

```
int My_allocation_using_function(
    int I, int p, int *speeds, int n);
algorithm Nbody(int n, int p, int speeds[p]) {
  coord I=p;
  node {
    I>=0: bench*(n*
       My_allocation_using_function(I, p, speeds, n));
  } ;
  …
};
```

In this performance model, to calculate the volume of computations in the **`node`** declaration performed by the processor with coordinate **`I`**, a function must be used (a user-defined function `My_allocation_using_function` is used in this case) which calculates and returns the number of bodies allocated to the processor **`I`** proportional to its speed.

3. For the function call to **`HMPI_Group_auto_create`**, the application programmers supply values for the input parameters in the parameter list to the performance model. The output parameters are ignored.

After the call to the function **`HMPI_Group_auto_create`**, the output parameters, namely, the number of processes in each dimension of the processor arrangement can be obtained by using the HMPI group accessor function **`HMPI_Group_topology`** and the relative performances of the processors can be obtained by using the HMPI accessor function **`HMPI_Group_performances`**. All the members of the group then use the optimal performances to distribute computations such that the volumes of computations are proportional to their performances. This is followed by execution of the algorithm by the members of the group.

The function **`HMPI_Group_auto_create`** evaluates all the possible process arrangements. For each process arrangement, the function call **`HMPI_Timeof`** is used to estimate the time of execution of the algorithm. The estimation is calculated based on the performance model of the parallel algorithm and the model of the executing network of computers. The function **`HMPI_Group_auto_create`** returns the process arrangement that results in the least estimated time of execution of the algorithm. As discussed in Section 3.4, the function call **`HMPI_Timeof`** invokes the mapping algorithms of the HMPI runtime system to select such a mapping that is estimated to ensure the fastest execution of the parallel algorithm. During the execution of the mapping algorithm, the HMPI runtime system searches for some approximate solution that can be found in some reasonable interval of time by probation of a subset of all possible mappings. During the execution of the mapping algorithm at the preliminary step, the HMPI runtime system re-orders the set of processors in accordance with the volume of computations to be performed by the processors, so that the most loaded processor comes first. The mapping algorithm thus re-orders processors in a non-increasing order of their speeds along each dimension of the processor arrangement. Beaumont *et al.* [BBP+01] show that the optimal mapping is one of the possible non-increasing arrangements where processors are arranged in a non-increasing order of their speed along each row and along each column of the 2D processor grid arrangement. The function **`HMPI_Group_auto_create`** thus internally invokes mapping algorithms that use this heuristic, which is to arrange the processors in a non-increasing order of their speed along each row and along each column of the 2D processor grid arrangement, in order to find the optimal number of processes that can execute the parallel application on HNOCs. The heuristics used may not be the best possible heuristics in most particular cases.

The pseudo-code of our research implementation of the function **`HMPI_Group_auto_create`** is shown below:

```
int i, pa, *opt_a, *a;
double t, T, *speeds, *opt_speeds;
void *model_parameters;
// Parent of the group
if (!HMPI_Is_free()) {
   int p = HMPI_Get_number_of_free_processes() + 1;
   GetProcessSpeeds(speeds);
   GenerateProcessArrangements(p, a, &pa);
   for (i=0, T=DBL_MAX; i<pa; i++) {
      // Estimate the time of execution for
      // process arrangement ai
      FillModelParameters(perf_model, &a[i], speeds,
                          model_parameters);
     t = HMPI_Timeof(perf_model, model_parameters);
     if (t < T) {
        T = t;
        opt_a = &a[i];
```

```
            opt_speeds = speeds;
         }
      }
      FillModelParameters(perf_model, opt_a, opt_speeds,
                          model_parameters);
      HMPI_Group_create(gid, perf_model, model_parameters);
      return HMPI_SUCCESS;
   }
   // Processes that are not members of any group
   HMPI_Group_create(gid, perf_model, NULL);
   return HMPI_SUCCESS;
```

The function **`HMPI_Group_auto_create`** first computes the number of processes available for computation. This includes the parent and all the processes, which are not members of any HMPI group. The function

```
HMPI_Get_number_of_free_processes()
```

returns the number of processes, which are not members of any HMPI group. This number is incremented by one to account for the parent of the group.

The function *`GetProcessSpeeds`* returns the relative performances of the processes that are available for computation.

Then the function *`GenerateProcessArrangements`* generates all the possible process arrangements. For example, if the number of processes available for computation is 9 and the topology (given by the performance model definition) is a 2-D process grid arrangement **`(p,q)`**, where **`p`** and **`q`** are the number of processes along the row and along the column of the process grid respectively, this function returns the following process arrangements:

```
(1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7) (1,8) (1,9)
              (2,1) (2,2) (2,3) (2,4)
                 (3,1) (3,2) (3,3)
                    (4,1) (4,2)
```

```
(5,1)
(6,1)
(7,1)
(8,1)
(9,1)
```

For each process arrangement generated above, the time of execution of the algorithm is estimated using the function `HMPI_Timeof` discussed in Section 3.4. The process arrangement that results in the least estimated time of execution is returned along with the relative performances of the processes in this process arrangement.

Complexity. Assume that the number of processes available for computation is `p` and the number of computers in the executing network is `M`. For each process arrangement generated, the time of execution of the algorithm is estimated using `HMPI_Timeof`. Assume there are `pa` number of process arrangements and the set of process arrangements is represented by `a`.

The overhead of the estimation of time of execution of the algorithm for each process arrangement is a product of two terms. The first term is the number of mappings probated. This is equal to the number of computers of the executing network multiplied by the number of processes in the process arrangement (Refer to Section 3.2.1.2). The second term is the overhead associated with estimation of time of execution of the algorithm for each such mapping. This is dependent on the code written by the user in the **scheme** declaration of the performance model definition to model how exactly the processes interact during the execution of the algorithm.

The calculation of the total overhead involved in a call to the function `HMPI_Group_auto_create` becomes complicated if this overhead associated with the estimation of execution time of the algorithm is a function of the number of processes involved in the execution of the algorithm and the size of the problem. To simplify the calculation of the total overhead, we assume the time of estimation of execution time of the algorithm for each mapping to be a function of the size of the problem $\mathrm{F(n)}$. We assume this dependency on the size of the problem to be the same for every process arrangement. We also assume only two-dimensional process arrangements.

The total overhead involved in a call to the function **`HMPI_Group_auto_create`** is

$$= \sum_{i=1}^{\mathrm{pa}} \left(\text{Estimation of execution time for process arrangement } \mathrm{a_i}\right)$$
$$= \sum_{i=1}^{\mathrm{pa}} \left(\mathrm{M} \times \left(\text{number of processes in process arrangement } \mathrm{a_i}\right) \times \mathrm{F(n)}\right)$$
$$= \mathrm{M} \times \mathrm{F(n)} \times \sum_{i=1}^{\mathrm{pa}} \left(\text{number of processes in process arrangement } \mathrm{a_i}\right)$$

Now consider for example **p**=5, then the possible two-dimensional process arrangements are {(1,1),(1,2),(1,3),(1,4),(1,5),(2,1),(2,2),(3,1),(4,1),(5,1)}, the term $\sum_{i=1}^{\mathrm{pa}} \left(\text{number of processes in process arrangement } \mathrm{a_i}\right)$ is equal to 1×1 + 1×2 + 1×3 + 1×4 + 1×5 + 2×1 + 2×2 + 3×1 + 4×1 + 5×1, which can be rearranged as (1+2+3+4+5) + 2×(1+5/2) + 3×(5/3) + 4×(5/4) + 5×(5/5).

Thus the total overhead is equal to

$$= M \times F(n) \times \left\{ (1+2+3+\cdots+p) + 2\times\left(1+2+3+\cdots+\frac{p}{2}\right) + 3\times\left(1+2+3+\cdots+\frac{p}{3}\right) + \cdots + p\times\left(\frac{p}{p}\right) \right\}$$

$$= M \times F(n) \times \left\{ \left(\frac{p\times(p+1)}{2}\right) + 2\times\left(\frac{\frac{p}{2}\times(\frac{p}{2}+1)}{2}\right) + 3\times\left(\frac{\frac{p}{3}\times(\frac{p}{3}+1)}{2}\right) + \cdots + p\times 1 \right\}$$

$$= M \times F(n) \times \frac{p}{2} \times \left\{ (p+1) + \left(\frac{p}{2}+1\right) + \left(\frac{p}{3}+1\right) + \cdots + \left(\frac{p}{p}+1\right) \right\}$$

$$= M \times F(n) \times \frac{p}{2} \times \left\{ p\times\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{p}\right) + (1+1+\cdots+1) \right\}$$

$$= O(M \times F(n) \times \frac{p}{2} \times \{p\times(\log_e(p)) + (1+1+\cdots+1)\})$$

$$= O(M \times F(n) \times \frac{p}{2} \times \{p\times(\log_e(p)) + p\})$$

$$= O(M \times F(n) \times p^2 \times \log_e p)$$

The sum for large **p** for the harmonic series $\left(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{p}\right) = O(\log_e p)$. The sum of the series 1+2+3+…+p is equal to $\frac{p\times(p+1)}{2}$. Assuming one process per processor configuration and that the number of computers in the executing network is **p**, the total overhead is $= O(p\times F(n)\times p^2 \times \log_e p) = O(F(n)\times p^3 \times \log_e p)$. For process arrangements with **d** dimensions in general, the total overhead can be calculated to be equal to $O(F(n)\times p^3 \times \log_e p \times d)$.

HMPI provides a variant of the operation **HMPI_Group_auto_create** that allows application programmers to supply heuristics that minimize the number of process arrangements evaluated. Its interface is shown below.

```
typedef int (*HMPI_Heuristic_function)(
     int np, int *dp, void *model_params, int param_count);
HMPI_Group_heuristic_auto_create (
     HMPI_Group* gid, const HMPI_Model* perf_model,
     HMPI_Heuristic_function hfunc,
     void* model_parameters)
```

Application programmers provide the heuristic function **hfunc**. The input parameter **`np`** is the number of dimensions in the process arrangement. The input parameter **`dp`** is an integer array of size **`np`** containing the number of processes in each dimension of the process arrangement. The input parameter **`model_params`** are the parameters supplied to the performance model. The input parameter **`param_count`** is the number of parameters in **`model_params`**.

The function **`HMPI_Group_heuristic_auto_create`** evaluates a process arrangement only if the heuristic function **hfunc** returns **true**. A simple heuristic function is shown below, which returns a value **true** if and only if the process arrangement is a square grid.

```
int Square_grid_only(
    int np, int *dp, void *model_params, int param_count){
    if ((np == 2) && (dp[0] == dp[1]))
       return true;
    return false;
}
```

The function **`HMPI_Group_heuristic_auto_create`** evaluates process arrangements that are square grids if this heuristic function is provided as an input. HMPI also provides predefined heuristic functions. The rest of the parameters to the function **`HMPI_Group_heuristic_auto_create`** have same meaning as those for the operation **`HMPI_Group_auto_create`**.

3.5.1 Experimental Results using HMPI_Group_auto_create

The example shown in Figures 3.13, 3.14, 3.15, and 3.16 illustrates the usage of the function **`HMPI_Group_auto_create`** on 1-D processor arrangements. The example used is an application multiplying matrix *A* and the transposition of matrix *B* on **p** interconnected heterogeneous processors, i.e., implementing matrix operation $\mathbf{C}=\mathbf{A}\times\mathbf{B}^{\mathbf{T}}$, where *A*, *B* are dense

n×**n** matrices. This application assumes one process per processor configuration and implements a naive heterogeneous algorithm shown in Figures 3.13 and 3.14. It can be summarized as follows:

- Each element in *C* is a square **r×r** block and the unit of computation is the computation of one block, i.e., a multiplication of **r×n** and **n×r** matrices.
- The *A*, *B*, and *C* matrices are identically partitioned into **p** horizontal slices. There is one-to-one mapping between these slices and the processors. Each processor is responsible for computing its C slice.

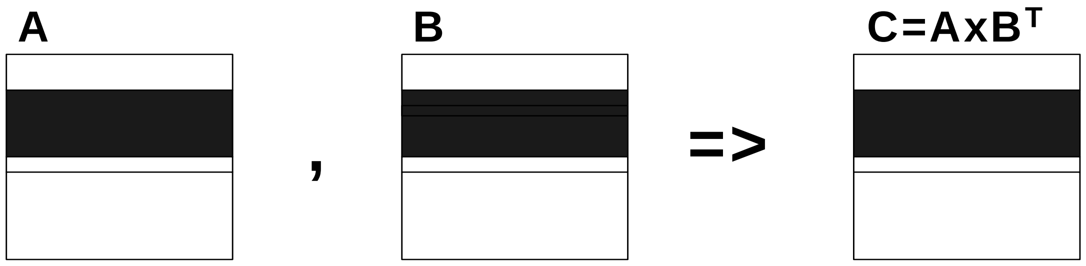


Figure 3.13: Matrix operation C=A×B^T with matrices A, B, and C unevenly partitioned in one dimension. The slices mapped onto a single processor are shaded in black.

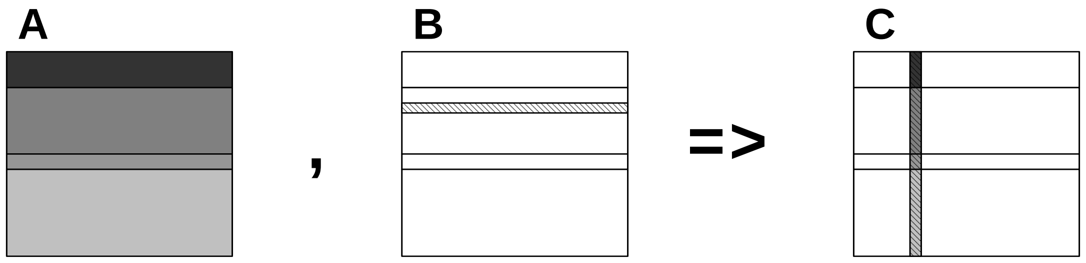


Figure 3.14: One step of parallel multiplication of matrices A and B. The pivot row of matrix B (shown slashed) is first broadcast to all processors. Then each processor computes, in parallel with the others, its part of the corresponding column of the resulting matrix C.

- At each step, a row of matrix *B* (the pivot row), representing a column of blocks of matrix B^T, is communicated (broadcast) vertically; and all processors compute the corresponding column of *C* in parallel.
- Because all *C* elements require the same amount of arithmetic operations, each processor executes an amount of work proportional to the number of blocks that are allocated to it, hence, proportional to the area of its slice. Therefore, to balance the load of the processor, the area of the slice mapped to each processor is proportional to it speed.
- Communication overheads may exceed gains due to parallel execution of computations. Therefore, there exists some optimal subset of available processors to perform the matrix multiplication. The function **`HMPI_Group_auto_create`** detects this optimal subset.

The definition of **`ParallelAxBT`** given in Figure 3.15 describes the performance model of this heterogeneous algorithm.

The performance model **`ParallelAxBT`** describing the algorithm has 5 parameters. Parameter **`n`** is the size of square matrices *A*, *B*, and *C*. It is assumed that the test code, used for the estimation of the speed of actual processors, multiplies **r**×**n** and **n**×**`t`** matrices, where **`t`** is small enough compared to **n** and supposed to be a multiple of **r**.

Parameters **`p`** and **`speeds`** are output parameters. They represent the number and the performances of processors in the optimal subset respectively.

Function **`Get_my_partition`** used in the **`node`** and **`link`** declarations is a set partitioning API, which is part of the Heterogeneous Data Partitioning Interface (HDPI) presented in Chapter 4. It partitions a set of **`n`** elements into **`p`** disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that

```
int Get_my_partition(int i, int p, const int *speeds, int n);
algorithm ParallelAxBT(int n, int r, int t, int p,
                       int speeds[p]) {
   coord I=p;
   node {
     I>=0: bench*((n/r/t)*Get_my_partition(I, p, speeds, n));
   };
   link (J=p) {
     I!=J: length*(Get_my_partition(J, p, speeds, n)
                *n*sizeof(double)) [J]->[I]
   };
   parent [0];
   scheme {
      int i, j, PivotProcessor=0, PivotRow=0, d[p];
      Partition_unordered_set(p, 1, speeds, NULL, NULL, n,
                  NULL, -1, NULL, NULL, d);
      for (i = 0; i < (n/r); i++, PivotRow+=1) {
         if (PivotRow >= d[PivotProcessor]) {
            PivotProcessor++;
            PivotRow = 0;
         }
         for (j = 0; j < p; j++)
            if (j != PivotProcessor)
               ((100.00*r)/d[PivotProcessor]) %%
               [PivotProcessor]->[j];
         par (j = 0; j < p; j++)
            ((100.00*r)/n) %% [j];
      }
  };
};
```

Figure 3.15: Specification of the performance model of an algorithm of parallel matrix multiplication in the HMPI's performance definition language.

partition. It returns the number of elements in the **i**-th partition belonging to the **i**-th processor. The **node** declaration specifies the volume of computations to be performed by the **i**-th processor executing the algorithm. The unit of computation used to measure the volume is the computation of one element of the resulting matrix *C*. It is presumed that function **serialAxBT** used in **HMPI_Recon** to update the performance model of the executing network of computers just implements this elementary computation. The **link** declaration specifies that each processor will send its *B* slice to all other processors executing the algorithm.

```
int main() {
    int opt_p, *opt_speeds, *model_params, n, r, t, nd, **dp;
    int output_p, input_p[3]= {n, r, t};
    HMPI_Group gid;

    HMPI_Init(argc, argv);

    if ((HMPI_Is_member(HMPI_PROC_WORLD_GROUP)) {
       HMPI_Recon(&serialAxBT, input_p, 3, &output_p);
    }
    if (HMPI_Is_host()) {
       // The user fills in only the first three parameters
       // Parameters ‘p’ and ‘speeds’ returned by
       // the call to function HMPI_Group_auto_create
       model_params[0] = n;
       model_params[1] = r;
       model_params[2] = t;
    }
    if (HMPI_Is_host())
       HMPI_Group_auto_create (&gid, &HMPI_Model_parallelAxBT,
                              model_params)
    if (HMPI_Is_free())
       HMPI_Group_auto_create (&gid, &HMPI_Model_parallelAxBT,
                              NULL)
    if (HMPI_Is_member(&gid)){
       HMPI_Group_topology(&gid, &nd, dp);
       opt_p = (*dp)[0];
       HMPI_Group_performances(&gid, opt_speeds);
       // Distribute computations using the optimal speeds of
       // processes.
       // computations and communications are performed here
    }

    if (HMPI_Is_member(&gid))
    {
       HMPI_Group_free(&gid);
    }

    HMPI_Finalize(0);
}
```

Figure 3.16: The most principal fragments of the usage of function **`HMPI_Group_auto_create`** for detection of the optimal subset of processors to execute the parallel matrix multiplication and creation of the corresponding optimal group of processes (one process per processor configuration is assumed).

The **`scheme`** declaration specifies **`n/r`** successive steps of the algorithm. At each step, the processor **`PivotProcessor`**, which hold the pivot row, sends it to rest of the processors thus executing **`(r/d[PivotProcessor])×100`** percent of total data transfer through the corresponding data link. Then, all processors compute the corresponding column of blocks of matrix *C* in parallel, each thus executing **`(r/n)×100`** percent of the total volume of computation to be performed by the processor.

Function **`Partition_unordered_set`** used in the **`scheme`** declaration is a set partitioning API, which is part of the Heterogeneous Data Partitioning Interface (HDPI) presented in Chapter 4. It is used to partition a set of size **`n`** into **`p`** disjoint subsets on a linear array of **`p`** processors such that the number of elements in each subset is proportional to the speed of the processor owning that subset.

The most principal fragments of rest of the code of the parallel application are shown in the **`main`** function in Figure 3.16.

HMPI runtime system is initialized using operation **`HMPI_Init`**. Then, operation **`HMPI_Recon`** updates the estimation of performances of processors using some serial multiplication of test matrices using function **`serialAxBT.`**

This is followed by the creation of a group of processes using operation **`HMPI_Group_auto_create`**. Users specify only the first three model parameters to the performance model and ignore the return parameters specifying the number of processes to be involved in executing the algorithm and their performances. This function calculates the optimal number of actual processes to be involved in the parallel matrix multiplication and their performances. After the execution of the function **`HMPI_Group_auto_create`**, the optimal number of actual processes **`opt_p`** is obtained by using the HMPI group accessor function **`HMPI_Group_topology`** and their performances **`opt_speeds`** are obtained by using the HMPI group accessor function **`HMPI_Group_peformances`**.

Name (Number of Processors)	**Architecture**	**cpu MHz**	**Total Main Memory (mBytes)**	**Cache (kBytes)**
afflatus(1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor supporting HT[†] technology	2867	2048	1024
aries2(1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor	2457	512	1024
pg1cluster01 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512
pg1cluster02 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512
pg1cluster03 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512
maxft (1)	Linux 2.6.5-1.358 Pentium III	731	128	256
zaphod (1)	Linux 2.4.18-14	497	128	512
csultra01 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra02 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra03 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra04 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra05 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra06 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048

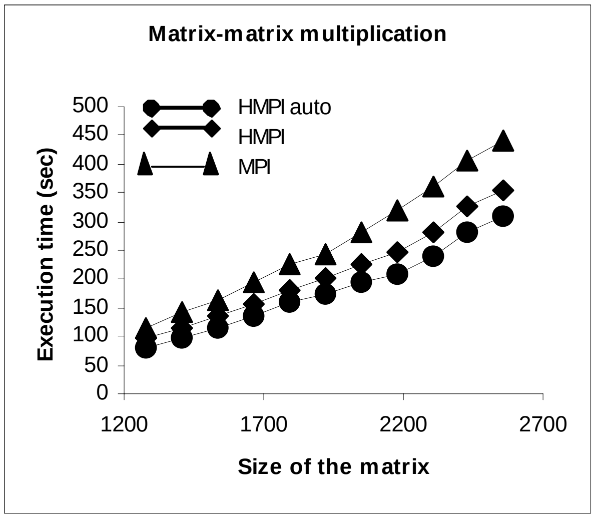
Table 3.6: Specifications of the sixteen heterogeneous processors used for the parallel matrix multiplication (only one process is run per processor).

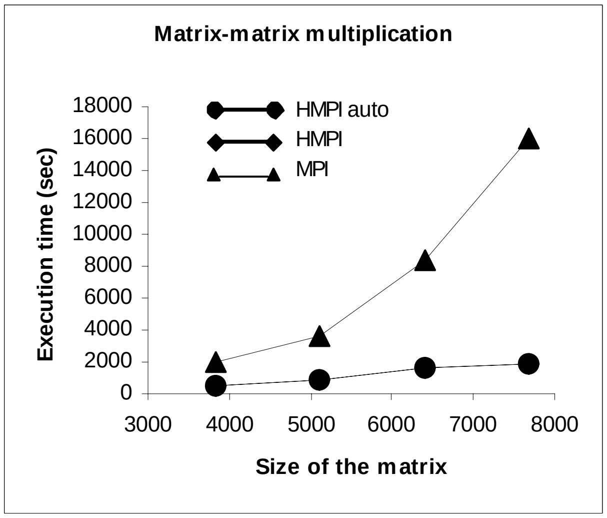
The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **`HMPI_Group_free`** and the finalization of HMPI runtime system using operation **`HMPI_Finalize`**.

Size of matrix (**n**)	HMPI application using **`HMPI_Group_auto_create`**		HMPI application using **`HMPI_Group_create`**	MPI
	Optimal number of processors (**p**)	Execution time (sec)	Execution Time (sec) (**p**=16)	Execution Time (sec) (**p**=16)
1280	7/8	79	98	114
1408	7/7	98	114	141
1536	7/7	116	134	162
1664	7/7	135	155	194
1792	7/8	160	179	224
1920	7/8	173	203	244
2048	7/7	193	225	282
2176	7/7	208	247	321
2304	8	240	280	360
2432	16	325	280	406
2560	16	353	308	441
3840	16	558	545	1979
5120	16	832	847	3584
6400	16	1593	1574	8385
7680	16	1820	1815	16024

Table 3.7: Comparison of execution times of the parallel matrix multiplication. There are a total of 16 processes available for computation in the parallel matrix multiplication. For each problem size, the HMPI application using **`HMPI_Group_auto_create`** finds the optimal number of processes whereas the HMPI application using **`HMPI_Group_create`** and the MPI application use all the available 16 processes. The MPI application does not take into account the speeds of the processors and the latencies and the bandwidths of the communication links between them.

A heterogeneous local network of 16 different FreeBSD, Solaris, and Linux workstations shown in Table 3.6 is used in the experiments. The computers used in the experiments are connected to communication network, which is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The experimental results are obtained





(a) (b)

Figure 3.17: Results obtained using the heterogeneous network of computers shown in Table 3.6. The results for '**HMPI auto**' are for the HMPI application using the function **`HMPI_Group_auto_create`**. The results for '**HMPI**' are for the HMPI application using the function **`HMPI_Group_auto_create`** and '**MPI**' are for the MPI application using all the available 16 processes. (a) Comparison of execution times of Matrix-matrix multiplication using horizontal striped partitioning of matrices for the sizes of the matrix in the range 1000-3000. (b) Comparison of execution times of Matrix-matrix multiplication using horizontal striped partitioning of matrices for the sizes of the matrix in the range 3000-8000.

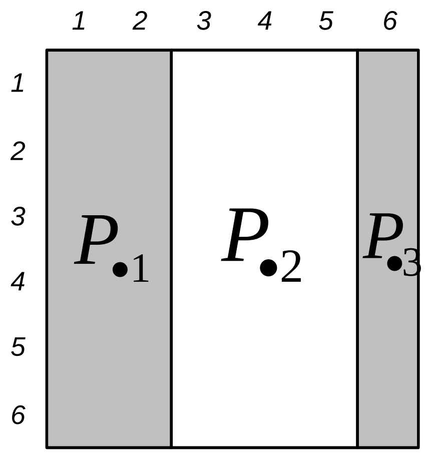
by averaging the execution times over a number of experiments. Table 3.7 and Figure 3.17 show the experimental results using the parallel matrix multiplication for different matrix sizes.

It can be seen that for problem sizes ranging from 1280 to 2304, the optimal number of processes detected by the function **`HMPI_Group_auto_create`** is 7 or 8 depending on the size of the matrix and the current performance demonstrated by different processors. For problem sizes beyond 2304, the optimal number of processes detected is 16, which is equal to the total number of parallel processes available for computation. It can also be seen that when the optimal number of processes detected by the function **`HMPI_Group_auto_create`** is equal to the total number of parallel processes available for computation, the execution times of the

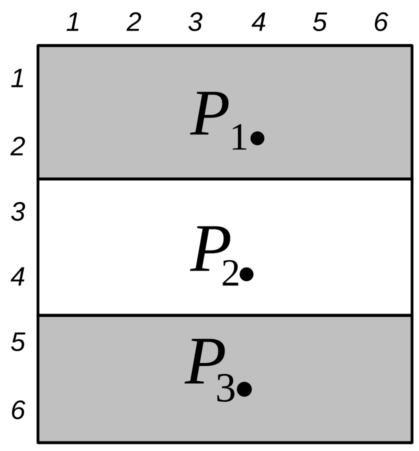
HMPI application using the function **`HMPI_Group_auto_create`** are almost the same as the execution times of the HMPI application using the function **`HMPI_Group_create`**, which creates a group consisting of all the parallel processes available for computation. This is to be expected because the function **`HMPI_Group_auto_create`** essentially detects the optimal number of processes and calls the function **`HMPI_Group_create`** providing the optimal number of processes as an input parameter. Thus the function **`HMPI_Group_auto_create`** by choosing the optimal number of processes to be involved in executing the application automatically removes some nodes from the computation when their participation degrades performance or when their participation does not affect the performance. The MPI application performs very poorly on this network because it does not take into account the speeds of the processors and the latencies and the bandwidths of the communication links between them.

The example described above illustrates the usage of the function **`HMPI_Group_auto_create`** to create a group of processes with optimal number of processes arranged linearly to solve the problem of parallel matrix-matrix multiplication. Hence the example demonstrates mainly the utility of the function **`HMPI_Group_auto_create`** on 1-D processor arrangements. However this function can be used for different arrangements of processors and its significance is demonstrated below on parallel matrix-matrix multiplication employing heterogeneous 2D block cyclic distribution.

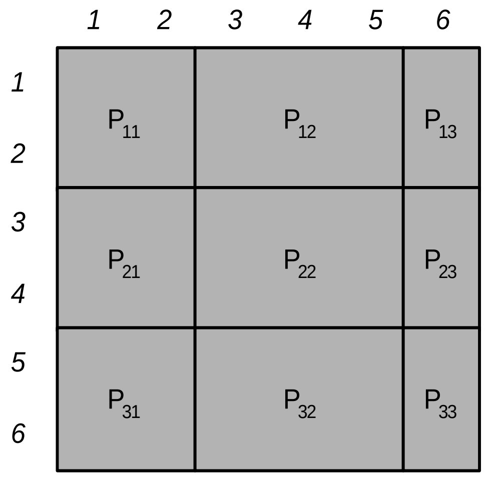
Consider the problem of parallel matrix multiplication (MM) on HNOCs. The algorithm of execution of the matrix operation $C=A\times B$ on a HNOC is obtained by modification of the ScaLAPACK [CDD+96] 2D block-cyclic MM algorithm. The modification is that the Cartesian heterogeneous 2D block-cyclic data distribution (shown in Figure 3.18) is used instead of the standard homogeneous data distribution. Thus, the heterogeneous algorithm of multiplication of



(a) Partition between processor columns.



(b) Partition between processor rows.



(c) Final Partition.

Figure 3.18: Example of two-step Cartesian distribution of a 6×6 generalized block over a 3×3 processor grid. The relative speed of processors is given by matrix $s=\begin{pmatrix}0.11 & 0.17 & 0.05\\ 0.17 & 0.09 & 0.08\\ 0.05 & 0.25 & 0.03\end{pmatrix}$. (a) At the first step, the 6×6 square is distributed in a one-dimensional block fashion over processors columns of the 3×3 processor grid in proportion $0.33:0.51:0.16 \approx 2:3:1$. (b) At the second step, the 6×6 square is distributed in a one-dimensional block fashion over processors rows of the 3×3 processor grid in proportion $0.33:0.34:0.33 \approx 2:2:2$. (c) Final partition.

two dense square (**n**×**r**)×(**n**×**r**) matrices A and B on an **p**×**q** grid of heterogeneous processors can be summarised as follows (graphically illustrated in Figure 3.19):

- Each element in A, B, and C is a square **r**×**r** block and the unit of computation is the updating of one block, i.e., a matrix multiplication of size **r**. Each matrix is partitioned into generalized blocks of the same size (**l_p**×**r**)×(**l_q**×**r**), where **p**≤**l_p**≤**n**, **q**≤**l_q**≤**n**. The generalized blocks are identically partitioned into **p**×**q** rectangles, each being assigned to a different processor. The area of each rectangle is proportional to the speed of the processor that stores the rectangle. The partitioning of a generalized block is performed as follows:
    - Each element in the generalized block is a square **r**×**r** block of matrix elements. The generalized block is an **l_p**×**l_q** rectangle of **r**×**r** blocks.
    - First, the generalized block **l_p**×**l_q** is partitioned into **q** vertical slices, so that the area of the *j*-th slice is proportional to $\sum_{i=1}^{p} s_{ij}$ (see Figure 3.18(a)). It is supposed that blocks of the *j*-th slice will be assigned to processors of the *j*-th column in the **p**×**q** processor grid. Thus, at this step, we balance the load between processor columns in the **p**×**q** processor grid, so that each processor column will store a vertical slice whose area is proportional to the total speed of its processors.
    - Then, the generalized **l_p**×**l_q** is partitioned into **p** horizontal slices, so that the area of the *i*-th slice is proportional to $\sum_{j=1}^{q} s_{ij}$ (see Figure 3.18(b)). It is supposed that blocks of the *i*-th slice will be assigned to processors of the *i*-th row in the **p**×**q** processor grid. Thus, at this step, we balance the load between processor

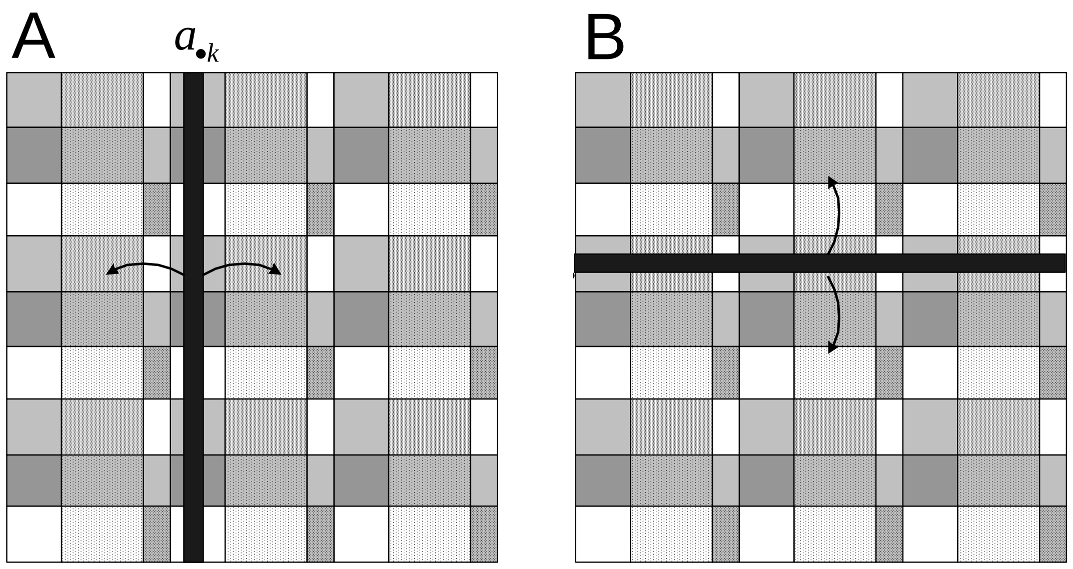


Figure 3.19: One step of the algorithm of parallel matrix-matrix multiplication based on heterogeneous two-dimensional block distribution of matrices A, B, and C. First, each r×r block of the pivot column $a_{\bullet k}$ of matrix A (shown shaded dark grey) is broadcast horizontally, and each r×r block of the pivot row $b_{k\bullet}$ of matrix B (shown shaded dark grey) is broadcast vertically.

rows in the **p**×**q** processor grid, so that each processor row will store a horizontal slice whose area is proportional to the total speed of its processors.

- At each step **k**,
    - Each **r**×**r** block a_{ik} of the pivot column of matrix A is sent horizontally from the processor, which stores this block, to **q**-1 processors (see Figure 3.19);
    - Each **r**×**r** block b_{kj} of the pivot row of matrix B is sent vertically from the processor, which stores this block, to **p**-1 processors (see Figure 3.19);
- Each processor updates its rectangle in the C matrix with one block from the pivot row and one block from the pivot column.

```
int Get_my_width(int i, int j, int p, int q, const int *speeds,
                 int td, int l_p, int l_q);
int Get_my_height(int i, int j, int p, int q, const int *speeds,
                 int td, int l_p, int l_q);
typedef struct {int I; int J;} Processor;
algorithm ParallelAxB(int n, int r, int l_p, int l_q,
                      int p, int q, int speeds[p×q])
{
  coord I=p, J=q;
  node {I>=0 && J>=0:
        bench*(Get_my_width(I, J, p, q, speeds, CARTESIAN, l_p, l_q)
             *(Get_my_height(I, J, p, q, speeds, CARTESIAN, l_p, l_q)
             *(n/l_p)*(n/l_q)*n);};
  link (K=p, L=q)
  {
    I>=0 && J>=0 && I!=K:
      length*(Get_my_width(I, J, p, q, speeds, CARTESIAN, l_p, l_q)
      *Get_my_height(I, J, p, q, speeds, CARTESIAN, l_p, l_q)
      *(n/l_p)*(n/l_q)*(r*r)*sizeof(double)) [I, J] -> [K, J];
    I>=0 && J>=0 && J!=L:
      length*(Get_my_width(I, J, p, q, speeds, CARTESIAN, l_p, l_q)
      *Get_my_height(I, J, p, q, speeds, CARTESIAN, l_p, l_q)
      *(n/l_p)*(n/l_q)*(r*r)*sizeof(double)) [I, J] -> [I, L];
  };
  parent[0,0];
  scheme {
    int k, *w, *h, *trow, *tcol;
    Processor Root, Receiver, Current;
    Partition_matrix_2d(p, q, 1, speeds, NULL, NULL, l_p, l_q, CARTESIAN,
                        w, h, trow, tcol, NULL, NULL);
    for(k = 0; k < n; k++) {
      int Acolumn = k%l_q, Arow;
      int Brow = k%l_p, Bcolumn;
      par(Arow = 0; Arow < l_p; )
      {
        Get_matrix_processor(Arow, Acolumn, p, q, w, h, trow, tcol, &Root);
        par(Receiver.J = 0; Receiver.J < q; Receiver.J++)
          if(Root.J != Receiver.J)
            (100.00/(w[Root.J]*(n/l_q)))
            %% [Root.I, Root.J] -> [Root.I, Receiver.J];
        Arow += h[Root.I];
      }
      par(Bcolumn = 0; Bcolumn < l_q; )
      {
        Get_matrix_processor(Brow, Bcolumn, p, q, w, h, trow, tcol, &Root);
        par(Receiver.I = 0; Receiver.I < p; Receiver.I++)
          if(Root.I != Receiver.I)
            (100.00/((h[Root.I])*(n/l_p)))
            %% [Root.I, Root.J] -> [Receiver.I, Root.J];
        Bcolumn += w[Root.J];
      }
      par(Current.I = 0; Current.I < p; Current.I++)
        par(Current.J = 0; Current.J < q; Current.J++)
          (100.00/n) %% [Current.I, Current.J];
    }
  };
};
```

Figure 3.20: Specification of the performance model of the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices in the HMPI's performance definition language.

The definition of **ParallelAxB** given in Figure 3.20 describes the performance model of this heterogeneous algorithm.

The performance model **ParallelAxB** describing the algorithm has 6 parameters. Parameter **r** specifies the size of a square block of matrix elements, the updating of which is the unit of computation of the algorithm. Parameter **n** is the size of square matrices *A*, *B*, and *C* measured in **r**×**r** blocks. Parameters **l_p**, and **l_q** are the sizes of the generalized block along the row and along the column and are also measured in **r**×**r** blocks.

Parameters **p**, **q**, and **speeds** are output parameters. They represent the number of processes along the row and the column in the process grid arrangement and the performances of processors in the optimal subset respectively.

The function **Get_my_width** and **Get_my_heigth** used in the **node** and **link** declarations and the function **Partition_matrix_2d** used in the **scheme** declaration is a matrix partitioning API, which is part of the Heterogeneous Data Partitioning Interface (HDPI) discussed in Chapter 4.

The function **Partition_matrix_2d** is used to partition a matrix of size **l_p×l_q** into **p×q** disjoint rectangles on a 2D processor grid arrangement **(p,q)** such that the area of each rectangle is proportional to the speed of the processor owning that rectangle.

The function **Get_my_width(I,J,…)** returns the width of the **J**-th rectangle belonging to the processor whose column index in the process grid is **J** (see Figure 3.18(a)). All widths are measured in **r**×**r** blocks. The function **Get_my_height(I,J,…)** returns the width of the **I**-th rectangle belonging to the processor whose row index in the process grid is **I** (see Figure 3.18(b)). All heights are measured in **r**×**r** blocks.

The **`coord`** declaration introduces 2 coordinate variables, **`I`** ranging from **`0`** to **`p-1`**, and **`J`** ranging from **`0`** to **`q-1`**.

The **`node`** declaration associates the abstract processors with this coordinate system to form a **`p`**×**`q`** grid. It also describes the absolute volume of computation to be performed by each of the processors. As a unit of measure, the volume of computation performed by the code multiplying two **`r`**×**`r`** matrices is used. At each step of the algorithm, abstract processor P_{IJ} updates $(w_{IJ} \times h_{IJ}) \times n_g$ **`r`**×**`r`** blocks, where w_{IJ}, h_{IJ} are the width and height of the rectangle of a generalised block assigned to processor P_{IJ}, and n_g is the total number of generalised blocks. As computations during the updating of one **`r`**×**`r`** block mainly fall into the multiplication of two **`r`**×**`r`** blocks, the volume of computations performed by the processor P_{IJ} at each step of the algorithm will be approximately $(w_{IJ} \times h_{IJ}) \times n_g$ times larger than the volume of computations performed to multiply two **`r`**×**`r`** matrices. As w_{IJ} is given by **`w[J]`**, h_{IJ} is given by **`h[I]`**, n_g is given by **`(n/l_p)*(n/l_q)`**, and the total number of steps of the algorithm is given by **`n`**, the total volume of computation performed by abstract processor P_{IJ} will be **`w[J]*h[I]*(n/l_p)*(n/l_q)*n`** times bigger than the volume of computation performed by the code multiplying two **`r`**×**`r`** matrices.

The **`link`** declaration specifies the volumes of data to be transferred between the abstract processors during the execution of the algorithm. The first statement in this declaration describes communications related to matrix *A*. Obviously, abstract processors from the same column of the processor grid do not send each other elements of matrix *A*. Only abstract processors from the same row of the processor grid send each other elements of matrix *A*. Abstract processor P_{IJ} will send elements of matrix *A* to processor P_{KL} only if its rectangle R_{IJ} in a generalised block has horizontal neighbours of the rectangle R_{KL} assigned to processor P_{KL}. In that case, processor P_{IJ} will send all such neighbours to processor P_{KL}. Thus, in total processor P_{IJ} will send $N_{IJKL} \times n_g$ **`r`**×**`r`** blocks of matrix *A* to processor P_{KL}, where N_{IJKL} is the number of horizontal neighbours of rectangle R_{KL} in rectangle R_{IJ}, and n_g is the total number of generalised blocks. As N_{IJKL} is given by **`w[J]* h[I]`**, n_g is given by **`(n/l_p)*(n/l_q)`**, and the volume of data in one **`r`**×**`r`** block is given by **`(r*r)*sizeof(double)`**, the total volume of data transferred from processor P_{IJ} to processor P_{KL} will be given by **`w[J]*h[I]*(n/l_p)*(n/l_q)*(r*r)*sizeof(double)`**.

The second statement in the **`link`** declaration describes communications related to matrix *B*. Obviously, only abstract processors from the same column of the processor grid send each other elements of matrix *B*. In particular, processor P_{IJ} will send all its **`r`**×**`r`** blocks of matrix *B* to all other processors from column *J* of the processor grid. The total number of **`r`**×**`r`** blocks of matrix *B* assigned to processor P_{IJ} is given by **`w[J]*h[I]*(n/l_p)*(n/l_q)`**.

The **`scheme`** declaration describes **`n`** successive steps of the algorithm. At each step **`k`**,

- A row of **`r`**×**`r`** blocks of matrix *B* is communicated vertically. For each pair of abstract processors P_{IJ} and P_{KJ} involved in this communication, P_{IJ} sends a part of this row to P_{KJ}. The number of **`r`**×**`r`** blocks transferred from P_{IJ} to P_{KJ} will be $w_{IJ} \times \left(\frac{n}{l\_q}\right)$, where $\left(\frac{n}{l\_q}\right)$ is the number of generalised blocks along the row of **`r`**×**`r`** blocks. The total number of **`r`**×**`r`** blocks of matrix *B*, which processor P_{IJ} sends to processor P_{KJ}, is $(w_{IJ} \times h_{IJ}) \times \frac{n}{l\_p} \times \frac{n}{l\_q}$. Therefore,

$$\frac{w_{IJ} \times \left(\frac{n}{l\_q}\right)}{(w_{IJ} \times h_{IJ}) \times \frac{n}{l\_p} \times \frac{n}{l\_q}} \times 100 = \frac{1}{h_{IJ} \times \left(\frac{n}{l\_p}\right)} \times 100$$ percent of data that should be in total sent from processor P_{IJ} to processor P_{KJ} will be sent at the step. The first nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies this fact. The **`par`** algorithmic patterns are used to specify that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.

- A column of **`r`**×**`r`** blocks of matrix *A* is communicated horizontally. If processors P_{IJ} and P_{KL} are involved in this communication so that P_{IJ} sends a part of this column to P_{KL}, then the number of **`r`**×**`r`** blocks transferred from P_{IJ} to P_{KL} will be $H_{IJKL} \times \left(\frac{n}{l\_p}\right)$, where H_{IJKL} is the height of the rectangle area in a generalised block, which is communicated from P_{IJ} to P_{KL}, and $\left(\frac{n}{l\_p}\right)$ is the number of generalised blocks along the column of **`r`**×**`r`** blocks. The total number of **`r`**×**`r`** blocks of matrix *A*, which processor P_{IJ} sends to processor P_{KL}, is $N_{IJKL} \times \frac{n}{l\_p} \times \frac{n}{l\_q}$. Therefore,

$$\frac{H_{IJKL} \times \left(\frac{n}{l\_p}\right)}{N_{IJKL} \times \frac{n}{l\_p} \times \frac{n}{l\_q}} \times 100 = \frac{H_{IJKL} \times \left(\frac{n}{l\_p}\right)}{(H_{IJKL} \times w_{IJ}) \times \frac{n}{l\_p} \times \frac{n}{l\_q}} \times 100 = \frac{1}{w_{IJ} \times \left(\frac{n}{l\_q}\right)} \times 100$$

percent of data that should be in total sent from processor P_{IJ} to processor P_{KL} will be sent at the step. The second nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration specifies this fact. Again, we use the **`par`** algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.

- Each abstract processor updates each its **`r`**×**`r`** block of matrix *C* with one block from the pivot column and one block from the pivot row, so that each block c_{ij} $(i, j \in \{1, \ldots, n\})$ of matrix *C* will be updated, $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. The processor performs the same volume of computation at each step of the algorithm. Therefore, at each of **`n`** steps of the algorithm the processor will perform $\frac{100}{n}$ percent of the volume of computations it performs during the execution of the algorithm. The third nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies this fact. The **`par`** algorithmic patterns are used here to specify that all abstract processors perform their computations in parallel.

Function **`Get_matrix_processor`** is used in the **`scheme`** declaration to iterate over abstract processors that store the pivot row and the pivot column of **`r`**×**`r`** blocks. It returns in its last parameter the grid coordinates of the abstract processor storing the **`r`**×**`r`** block, whose coordinates in a generalised block of a matrix are specified by its first two parameters. This function is also a matrix partitioning API, which is part of the Heterogeneous Data Partitioning Interface (HDPI) discussed in Chapter 4.

The performance model **`ParallelAxB`** shown in the Figure 3.20 is applicable to the heterogeneous algorithm with **`CARTESIAN`** data distribution. However this model can be made generic and applicable for any type of distribution by adding an extra parameter to its parameter list (the type of distribution) and using heterogeneous data partitioning API (presented in Chapter 4) in the body of the performance model. This extra parameter is the type of data distribution such as **`COLUMN_BASED`** or **`ROW_BASED`** or **`CARTESIAN`** or **`RECURSIVE`**.

```
int main(int argc, char** argv) {
   int opt_p, opt_q, *opt_speeds, *model_params, nd, **dp;
   int output_p, input_p[2] = {n, r};
   int n, r, l_p, l_q;
   HMPI_Group gid;

   HMPI_Init(argc, argv);
   if (HMPI_Is_member(HMPI_PROC_WORLD_GROUP))
      HMPI_Recon(&rMxM, input_p, 1, &output_p);

   if (HMPI_Is_host()) {
      // The user fills in only the first four parameters
      // Parameters 'p', 'q', and 'speeds' returned by
      // the call to function HMPI_Group_auto_create
      model_params[0] = n;
      model_params[1] = r;
      model_params[2] = l_p;
      model_params[3] = l_q;
   }
   if (HMPI_Is_host())
      HMPI_Group_auto_create(&gid, &HMPI_Model_ParallelAxB,
                             model_params);
   if (HMPI_Is_free())
      HMPI_Group_auto_create(&gid, &HMPI_Model_ParallelAxB,
                             NULL);

   if (HMPI_Is_member(&gid)) {
      HMPI_Group_topology(&gid, &nd, dp);
      opt_p = (*dp)[0];
      opt_q = (*dp)[1];
      HMPI_Group_performances(&gid, opt_speeds);
      // computations and communications are performed here
      // using standard MPI routines.
   }
   if (HMPI_Is_member(&gid)) {
      HMPI_Group_free(&gid);
   }
   HMPI_Finalize(0);
}
```

Figure 3.21: The most principal fragments of the usage of function **`HMPI_Group_auto_create`** for detection of the optimal processor grid arrangement to execute the parallel matrix multiplication implementing the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices and creation of the corresponding optimal group of processes (one process per processor configuration is assumed).

```
algorithm ParallelAxB(int n, int r, int l_p, int l_q,
   int type_of_distribution, int p, int q, int speeds[p*q])
```

The most principal fragments of the rest code of the parallel application are shown in the **`main`** function in Figure 3.21.

HMPI runtime system is initialised using operation **`HMPI_Init`**. Then, operation **`HMPI_Recon`** updates the estimation of performances of processors using the serial multiplication of test matrices of size **`r`**×**`r`**. The computations performed by each processor mainly fall into the execution of calls to function **`rMxM`**.

This is followed by the creation of a group of processes using operation **`HMPI_Group_auto_create`**. Users specify only the first four model parameters to the performance model and ignore the return parameters specifying the number of processes in each dimension of the processor grid arrangement to be involved in executing the algorithm and their performances. This function detects the optimal grid arrangement of processes to be involved in the parallel matrix multiplication and their performances. After the execution of the function **`HMPI_Group_auto_create`**, the optimal grid arrangement of processes (**`opt_p,opt_q`**) is obtained by using the HMPI group accessor function **`HMPI_Group_topology`** and their performances **`opt_speeds`** are obtained by using the HMPI group accessor function **`HMPI_Group_peformances`**.

The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **`HMPI_Group_free`** and the finalization of HMPI runtime system using operation **`HMPI_Finalize`**.

A heterogeneous local network of 12 different FreeBSD, Solaris, and Linux workstations shown in Table 3.8 is used in the experiments. The computers used in the experiments are connected to communication network, which is based on 100 Mbit Ethernet with a switch

Name (Number of Processors)	**Architecture**	**cpu MHz**	**Total Main Memory (mBytes)**	**Cache (kBytes)**
afflatus(1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor supporting HT† technology	2867	2048	1024
aries2(1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor	2457	512	1024
pg1cluster01 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512
pg1cluster02 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512
pg1cluster03 (1)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512
csultra01 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra02 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra03 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra04 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048
csultra05 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048

Table 3.8: Specifications of the twelve heterogeneous processors used for the parallel matrix multiplication using heterogeneous 2D block cyclic distribution. pg1cluster01, pg1cluster02, and pg1cluster03 are all dual processor machines whereas the rest of them are all single processor machines. Only one processor on pg1cluster03 is used for the experiments. Only one process is run per processor.

enabling parallel communications between the computers. The experimental results are obtained by averaging the execution times over a number of experiments. Tables 3.10 and 3.12 show the experimental results using the parallel matrix multiplication using heterogeneous block-cyclic distribution for different matrix sizes.

Size of matrix (**n**)	MPI
	`(p,q)=(3,4)`
	Execution time (sec)
1536	248
3072	1252
4608	3447
6144	7692

Table 3.9: Execution times of the parallel matrix multiplication using homogeneous block cyclic distribution. There are a total of 12 processes available for computation. The values of **p**=3, **q**=4, **`r`**=32, **`l_p`**=1536, and **`l_q`** =1536 are used in the experiments.

Size of matrix (**n**)	HMPI application using **`HMPI_Group_auto_create`**	HMPI application using **`HMPI_Group_create`**					
		`(p,q)`					
	Optimal grid arrangement **`(p,q)=(3,4)`**	(1,12)	(2,6)	**(3,4)**	(4,3)	(6,2)	(12,1)
	Execution time (sec)	Execution time (sec)					
1536	35	86	47	**35**	42	48	87
3072	167	347	208	**164**	178	197	351
4608	452	843	651	**442**	507	543	813
6144	905	1723	1582	**875**	1286	1229	1536

Table 3.10: Comparison of execution times of the parallel matrix multiplication using heterogeneous block cyclic distribution. There are a total of 12 processes available for computation. For each problem size, the HMPI application using **`HMPI_Group_auto_create`** finds the optimal arrangement of processes in a grid. The values of **`r`**=32, **`l_p`**=1536, and **`l_q`** =1536 are used in the experiments.

Size of matrix (**n**)	MPI
	`(p,q)=(3,4)`
	Execution time (sec)
2304	622
4608	3914
6912	14220

Table 3.11: Execution times of the parallel matrix multiplication using homogeneous block cyclic distribution. There are a total of 12 processes available for computation. The values of **p**=3, **q**=4, **`r`**=32, **`l_p`**=2304, and **`l_q`** =2304 are used in the experiments.

Size of matrix (**n**)	HMPI application using **`HMPI_Group_auto_create`**	HMPI application using **`HMPI_Group_create`**					
		`(p,q)`					
	Optimal grid arrangement **`(p,q)=(4,3)`**	(1,12)	(2,6)	**(3,4)**	**(4,3)**	(6,2)	(12, 1)
	Execution time (sec)	Execution time (sec)					
2304	97	187	106	**83**	**80**	102	193
4608	530	875	643	**515**	**517**	557	815
6912	1855	2486	2182	**1800**	**1802**	2052	2236

Table 3.12: Comparison of execution times of the parallel matrix multiplication using heterogeneous block cyclic distribution. The values of **`r`**=32, **`l_p`**=2304, and **`l_q`** =2304 are used in the experiments.

For the values of **`r`**=32, **`l_p`**=1536, and **`l_q`**=1536 used in the experiments, the optimal grid arrangement of processes **`(p,q)`** detected by the function **`HMPI_Group_auto_create`** is **`(3,4)`**. For the optimal processor grid arrangement **`(3,4)`**, it can be seen that the execution

(p,q)	(1,2)	(1,3)	(1,4)	(1,6)	(1,8)	(1,9)	(1,12)	(2,1)	(2,2)	(2,3)	(2,4)	(2,6)	(3,1)	(3,2)	(3,3)	(3,4)	(4,1)	(4,2)	**(4,3)**	(6,1)	(6,2)	(8,1)	(9,1)	(12,1)
Estimated Execution time	7198	4798	3599	2052	1772	1772	1772	7198	3599	2043	1699	1699	4798	2068	1652	1651	3599	1678	**1617**	2052	1652	1772	1772	1772

Table 3.13: Estimated execution times of the parallel matrix multiplication using heterogeneous block cyclic distribution for all the possible processor grid arrangements such that **`((l_p/r)%(p), (l_q/r)%(q))`** are zero (that is the matrices are partitioned into an whole number of generalized blocks). There are a total of 12 processes available for computation. The values of **`r`**=32, **`l_p`**=2304, **`l_q`**=2304, and size of the matrix **`n×r`**=6912 are used in the experiments.

times of the HMPI application using the function **`HMPI_Group_auto_create`** are greater than the execution times of the HMPI application using the function **`HMPI_Group_create`** by some seconds. This is because the function **`HMPI_Group_auto_create`** tries to estimate the execution time of the parallel matrix-matrix multiplication for each possible processor grid arrangement and calls the function **`HMPI_Group_create`**, providing the optimal grid arrangement of processes as an input parameter. Thus this marginal difference in the execution times between the HMPI application using the function **`HMPI_Group_auto_create`** and the HMPI application using the function **`HMPI_Group_create`** is the extra time involved in detecting the optimal grid arrangement of processes by the function **`HMPI_Group_auto_create`**.

For the values of **`r`**=32, **`l_p`**=2304, and **`l_q`**=2304 used in the experiment**s,** the optimal grid arrangement of processes detected by the function **`HMPI_Group_auto_create`** is **`(4,3)`**. It can be seen from the execution times of the HMPI application using the function **`HMPI_Group_create`** that the optimal processor grid arrangements are **`(3,4)`**, and **`(4,3)`** from all the possible grid arrangements. For the values of **`r`**=32, **`l_p`**=2304, **`l_q`**=2304, and size of the matrix **`n×r`**=6912 used in the experiment**s,** the Table 3.13 shows the estimated execution times for all the possible processor grid arrangements.

The experimental results for the MPI application are shown in Tables 3.9 and 3.11. It performs very poorly on this network compared to the HMPI equivalent because it does not take into account the speeds of the processors and the latencies and the bandwidths of the communication links between them.

3.6 Model of HMPI Program

A typical HMPI application starts with the initialization of the HMPI runtime system using the operation

```
   HMPI_Init (int argc, char** argv)
```

where **`argc`** and **`argv`** are the same arguments, passed into the application, as the arguments to **`main`**. This routine must be called before any other HMPI routine and must be called once. This routine must be called by all the processes running in the HMPI application.

After the initialization, application programmers can call any other HMPI routines. In addition, MPI users can use normal MPI routines, with the exception of MPI initialization and finalization, including the standard group management and communicator management routines to create and free groups of MPI processes. However, they must use the predefined communication universe **`HMPI_COMM_WORLD`** of HMPI instead of **`MPI_COMM_WORLD`** of MPI.

The initialization of HMPI runtime system is typically followed by

- Updating of the estimation of the speeds of processors with **HMPI_Recon**;
- Finding the optimal values of the parameters of the parallel algorithm with **HMPI_Timeof**;
- Creation of a group of processes, which will perform the parallel algorithm, by using **HMPI_Group_create** or **HMPI_Group_auto_create**;
- Execution of the parallel algorithm by the members of the group. At this point, control is handed over to MPI. MPI and HMPI are interconnected by the operation **`HMPI_Get_comm`**, which returns an MPI communicator associated with communication group of MPI processes. Application programmers can use this communicator to call the standard MPI communication routines during the execution of the parallel algorithm. This communicator can safely be used in other MPI routines.
- Freeing the HMPI groups with **`HMPI_Group_free`**.
- Finalizing the HMPI runtime system by using operation

```
        HMPI_Finalize (int exitcode).
```

An HMPI application is like any other MPI application and can be deployed to run in any environment where MPI applications are used. HMPI applications can be run in environments where batch queuing and resource management systems are used. However HMPI uses its own

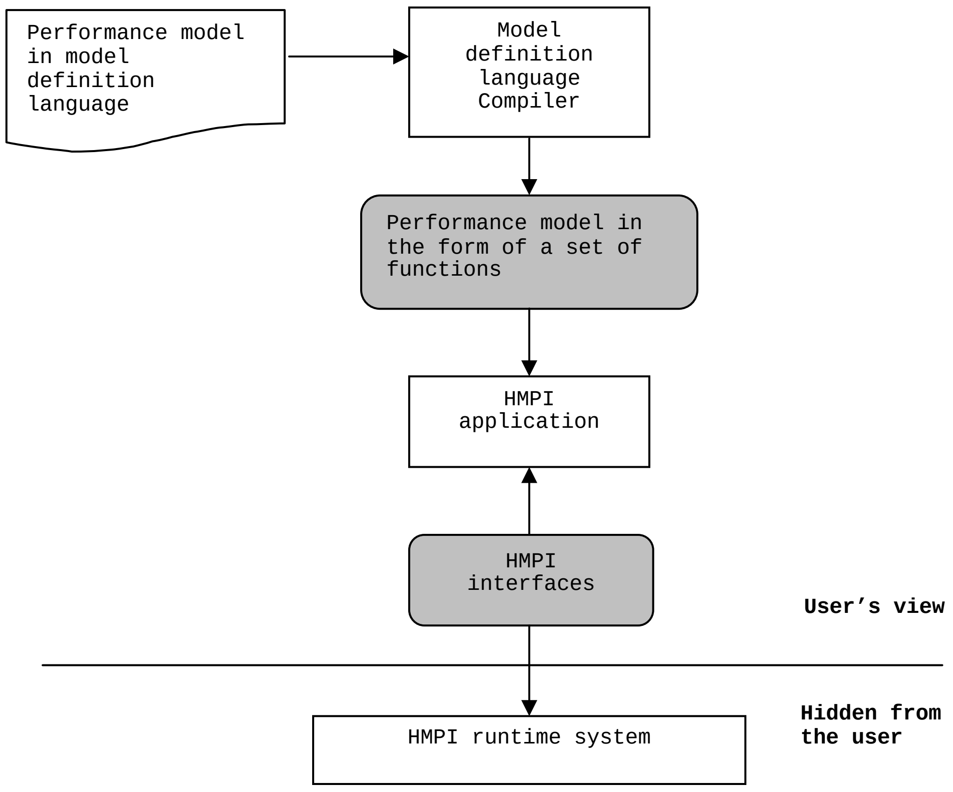

Performance model in model definition language
Model definition language Compiler
Performance model in the form of a set of functions
HMPI application
HMPI interfaces
User’s view
Hidden from the user
HMPI runtime system

Figure 3.22: Development process of an HMPI application. To build HMPI applications, an application programmer describes a performance model using the model definition language, compiles the performance model description into a set of functions, writes the application using the HMPI interfaces to create groups of processes to execute the parallel algorithm.

measurements and performance models of the underlying system for running parallel applications efficiently.

Note, that in general, the architecture of HMPI summarized in Figure 3.20 has similarities to the architectural framework of the CORBA specification [OMG98].

<u>3.7 Transformation of MPI to HMPI</u>

The section explains the steps involved in the transformation from an MPI program to an HMPI program.

```
int main(int argc, char **argv) {
    int i, x, y, l, m, t, n, r, me, p, *d;
    double val, *A, *B, *C, *temp;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    // Homogeneous data distribution
    Partition_unordered_set(p, 1, NULL, NULL, NULL, n,
             NULL, -1, NULL, NULL, d);

    // Execution of the algorithm by the members of MPI_COMM_WORLD
    for (i = 0; i < (n/r); i++) {
       int PivotProcessor = Get_set_processor(i, n/r, p, 0, d);
       MPI_Bcast(temp, n*r, MPI_DOUBLE, PivotProcessor,
                  *MPI_COMM_WORLD);
       for (x = 0; x < d[me]; x++)
          for (y = 0; y < (n/r); y++)
             for (l = 0; l < r; l++)
                for (m = 0; m < r; m++) {
                   for (val = 0, t = 0; t < r; t++) {
                       val += A[x*r*n + i*r + l*n + t]
                              *temp[y*r + t*r + m];
                   }
                   C[x*r*n + y*r + l*r + m] += val;
                }
    }
    MPI_Finalize();
}
```

Figure 3.23: The most relevant fragments of code of the MPI program implementing the parallel matrix-matrix multiplication algorithm shown in Figures 3.13 and 3.14.

Consider the example multiplying matrix *A* and the transposition of matrix *B* on **p** interconnected heterogeneous processors, i.e., implementing matrix operation $\mathbf{C}=\mathbf{A}\times\mathbf{B}^{\mathbf{T}}$, where *A*, *B* are dense **n**×**n** matrices. This application assumes one process per processor configuration and implements a naive heterogeneous algorithm shown in Figures 3.11 and 3.12. Figure 3.21 shows the MPI program. Figure 3.15 shows the performance model of the matrix-matrix multiplication algorithm and Figure 3.24 shows the HMPI program.

The straightforward transformations consist of one-to-one replacement of the MPI components by the HMPI counterparts. They are:

- **`HMPI_Init`** for **`MPI_Init`**, **`HMPI_Finalize`** for **`MPI_Finalize`**.
- HMPI pre-defined universe **`HMPI_Comm_world`** for MPI pre-defined universe **`MPI_Comm_world`**.
- HMPI group accessors **`HMPI_Group_rank`**, and **`HMPI_Group_size`** for MPI group accessors **`MPI_Group_rank`**, and **`MPI_Group_size`**.

There is absolutely no change in the code consisting of computations and communications of the parallel algorithm between an HMPI program and the MPI program. The MPI communicator used in this code can be replaced with the MPI communicator provided by the operation **`HMPI_Get_comm`** on the HMPI group of processes.

The other transformations are a bit involved and are outlined below in the order of increasing complexity:

- Determination of the speeds of the processors using **`HMPI_Recon`**.
- Creation of an HMPI group of processes that will execute the heterogeneous parallel algorithm using the operation **`HMPI_Group_create`**. The parameters to the performance model passed to this operation can be packed using the function **`HMPI_Pack_model_parameters`**.
- Destruction of an HMPI group once the execution of the algorithm is finished using the operation **`HMPI_Group_free`**. This is similar to the group destructor for an MPI group of processes **`MPI_Group_free`**.
- Description of the heterogeneous algorithm in the form of a performance model.

```
int main(int argc, char **argv) {
    int i, x, y, l, m, t, n, r, me, p, *d, nd, **dp, *opt_speeds, opt_p;
    int output_p, input_p[3] = {n, r, t};
    double val, *A, *B, *C, *temp;
    double *speeds;
    void *model_params;
    HMPI_Group gid;
    HMPI_Init(argc, argv);
    MPI_Comm_size(HMPI_COMM_WORLD, &p);
    // Estimation of speeds and data distribution using the speeds
    if ((HMPI_Is_member(HMPI_PROC_WORLD_GROUP))
       HMPI_Recon(&serialAxBT, input_p, 3, &output_p);
    HMPI_Group_performances(HMPI_COMM_WORLD_GROUP, &speeds);
    // Heterogeneous data distribution
    Partition_unordered_set(p, 1, speeds, NULL, NULL,
            n, NULL, -1, NULL, NULL, d);
    // HMPI Group creation
    if (HMPI_Is_host()) {
       model_params[0]=n;
       model_params[1]=r;
       model_params[2]=t;
       HMPI_Group_auto_create(&gid, &HMPI_Model_parallelAxBT, model_params);
    }
    if (HMPI_Is_free())
       HMPI_Group_auto_create(&gid, &HMPI_Model_parallelAxBT, NULL);
    // Execution of the algorithm by the members of the group
    me = HMPI_Group_rank(&gid);
    if (HMPI_Is_member(&gid)) {
       HMPI_Group_topology(&gid, &nd, dp);
       opt_p = (*dp)[0];
       HMPI_Group_performances(&gid, opt_speeds);
       Partition_unordered_set(opt_p, 1, opt_speeds, NULL, NULL, n,
                    NULL, -1, NULL, NULL, d);
       MPI_Comm mxm_comm = *(MPI_Comm*)HMPI_Get_comm(&gid);
       for (i = 0; i < (n/r); i++) {
          int PivotProcessor = Get_set_processor(i, n/r, p, 0, d);
          MPI_Bcast(temp, n*r, MPI_DOUBLE, PivotProcessor, mxm_comm);
          for (x = 0; x < d[me]; x++)
             for (y = 0; y < (n/r); y++)
                for (l = 0; l < r; l++)
                   for (m = 0; m < r; m++) {
                      for (val = 0, t = 0; t < r; t++)
                         val += A[x*r*n + i*r + l*n + t]
                                *temp[y*r + t*r + m];
                      C[x*r*n + y*r + l*r + m] += val;
             }
       }
    }
    // HMPI Group Destruction
    if (HMPI_Is_member(&gid)){
       HMPI_Group_free(&gid);
    }
    HMPI_Finalize(0);
}
```

Figure 3.24: The most relevant fragments of code of the HMPI program implementing the parallel matrix-matrix multiplication algorithm shown in Figures 3.13 and 3.14.

It can be seen that the most involved part in the transformation process is the design of the performance model.

The main constructs of the specification of the performance model definition language are briefly described here. The **coord** declaration specifies the arrangement of processes. The **node** declaration describes the total volume of computations to be performed by each of the processes in the group during the execution of the algorithm. The **link** declaration specifies the total volume of data to be transferred between each pair of processes in the group during the execution of the algorithm. The **scheme** declaration describes the order of execution of the computations and communications by the involved parallel processes in the group, that is, how exactly the processes interact during the execution of the algorithm.

The parameters to the performance model are mainly and usually the number of processes in each dimension of the process arrangement and data distribution parameters specifying how the data is distributed amongst the processes, and the amount of data that is transferred between the pair of processes. For example, if the mathematical objects used in the parallel algorithm are sets, the data distribution parameters are usually an array giving the number of elements in the set assigned to each processor proportional to the speed of the processor and an array giving the number of elements transferred between pairs of processors. If the mathematical objects used are matrices, the data distribution parameters are arrays giving the geometric dimensions of the partitions, which are rectangles assigned to each processor. If the mathematical objects used are graphs and trees, the data distribution parameters are arrays giving the number of nodes assigned to each processor and the edges that cross between pairs of processors.

It would appear that the description of the heterogeneous algorithm in the form of the performance model could be very complicated. However the user who has designed an MPI application has complete knowledge of the essential features of the parallel algorithm used in the MPI application. While designing the performance model, all that the user has to do is to explicitly specify these features in a parametric way. The specification of the performance model provides all the features to allow the user to specify all these features outlined previously in a general way without going into the nitty-gritty of the parallel algorithm.

The complexity of the performance model depends on how complex is the algorithm that the user has designed for the parallel application. The user can simplify the design of the performance model by ignoring some details of the parallel algorithm that have little or no influence on the performance of the parallel application. The performance model language offers all the features allowing the users to design all types of performance models ranging from the simplest to most complicated, and from not very accurate to very accurate for their parallel application. In some cases, a simple performance model can be designed that can accurately represent the essential features of the parallel algorithm used in their parallel applications. The specification of the performance model is comprehensive enough for expressing many scientific applications, as shown by the examples presented in this chapter and the scientific applications presented in Chapter 5 on HMPI application programming. At the same time it is expected to be improved based on the feedback from the scientific community using it.

An interesting topic is applications where different parallel algorithms are coupled. There are many ways of writing performance models and programming such applications in HMPI. If the application is composed of two algorithms that are loosely coupled, two different groups of different performance models executing the algorithms in parallel can be created. The HMPI runtime system will try to map the algorithms in such a way to ensure the best execution performance of the whole application. Alternatively, two different groups of different performance models executing the algorithms serially can be created (especially, in the case of strong data dependency). In the latter case, the first group must be destroyed before the second one is created to make all resources available when mapping each of the algorithms on the underlying hardware. If the two algorithms are tightly coupled, they can be described using a single performance model and executed on the same group.

Future work may involve the development of a tool that would automatically make some straightforward transformations to an MPI program to convert it into an HMPI program. The tool could be as simple as a script or a preprocessor that generates a basic working version of an HMPI program from an input MPI program. All that the application programmer will have to do is to design a performance model and input this performance model and MPI programs to the compiler or preprocessor. Hooks can be provided that allow the application programmers to specify the different stages of an MPI program that would aid the transformation process. These are the following:

- MPI initialization,
- Data distribution,
- Execution of the algorithm by the processes of **`MPI_COMM_WORLD`**, and
- MPI finalization.

Based on this information, a basic working version of a HMPI program can be generated from the performance model provided by the application programmer and the static program analysis of the MPI program. The basic working version would contain the following:

- HMPI initialization replacing the MPI initialization,
- Data distribution using the speeds of the processors. This step uses the Heterogeneous Data Partitioning Interface (HDPI) presented in Chapter 4. The application programmer

must dynamically update the processor speeds at runtime using **`HMPI_Recon`** before distributing the data.

- Creation of a HMPI group of processes. The call to the HMPI group creation function **`HMPI_Group_create`** is inserted. The handle to the performance model in the group creation function is generated by compiling the performance model provided as input by the application programmer. The application programmer will have to fill in the model parameters using the function **`HMPI_Pack_model_parameters`**.
- Execution of the algorithm by the processes of MPI communicator associated with the HMPI group of processes. This piece of code is similar to the MPI code except that the MPI communicator **`MPI_COMM_WORLD`** is replaced by the MPI communicator associated with the HMPI group of processes
- Destruction of the HMPI group of processes. The call to the group destruction function **`HMPI_Group_free`** is inserted, and
- HMPI finalization replacing the MPI finalization.

3.8 A Research Implementation of HMPI

The first version of a research implementation of HMPI is available from our homepage http://cs-www.ucd.ie/~hmpi.

The HMPI programming system includes the following components:

- A compiler to compile the performance model definitions.
- Run-time support system (RTSS).

- Library consisting of extensions to MPI and data partitioning API called HDPI (presented in Chapter 4). Currently this API has bindings to only ANSI C. Future work will involve design of bindings to C++, FORTRAN, and Java.
- Command-line user interface.

The compiler compiles the description of a performance model to generate a set of functions with calls to functions of RTSS. RTSS manages processes, constituting the parallel program, and provides communications. It encapsulates a particular communication platform (currently, a subset of MPI) ensuring platform-independence of the rest of system components. The command-line interface consists of tools for virtual parallel machine (VPM) management and execution of HMPI applications on the VPM.

Appendix B contains the HMPI Programmer's guide and installation guide. The Programmer's guide presents the HMPI library consisting of extensions to MPI and Heterogeneous Data Partitioning Interface (HPDI presented in Chapter 4) and the command-line interface to manage virtual parallel machine (VPM) and execute HMPI applications on VPM. The installation guide presents instructions to install HMPI on UNIX platforms (currently HMPI is available only for UNIX platforms).

In the following sections, we explain how to describe a VPM followed by the structure of the topology file, representing the model of the executing network of computers, generated during the creation of a VPM. We then present the model of HMPI program. This is followed by explanation of the synchronization functions and functions of RTSS performing process management tasks ensuring proper execution of HMPI applications.

3.8.1 Virtual parallel machine

The description of a virtual parallel machine (VPM) on which the HMPI applications are executed is provided in a VPM description file. A VPM description file consists of lines of two kinds. Lines starting with symbol '#' are treated as comments. All other lines should be of the following format:

```
<name> <number_of_processes> [number_of_processors]
```

where `<name>` is the name of the corresponding computer as it appears in the system '`/etc/hosts`' file, `<number_of_processes>` is the number of processes to run on the computer, and `[number_of_processors]` is the number of processors present on the computer (this is not mandatory). The host computer must go first in the file.

For example, the following file describes VPM consisting of three computers (`alpha`, `beta`, and `gamma`), five processes running on each computer, and the host computer is `alpha`:

```
alpha 5
beta 5
gamma 5
```

The VPM can be created, opened, closed or queried using VPM management tools. During the creation of the VPM, a topology file is generated that contains information on the model of the executing network of computers. The topology file for the VPM described previously is shown below:

```
parallel(0.49, 0.97) c62377 c967039 c801049
#alpha
s2 p6667 n5 serial c2285064 c107326590 c99523787
#beta
s2 p5556 n5 serial c1312665 c80473880 c98430419
#gamma
s2 p5556 n5 serial c1956722 c78885862 c99667528
```

In the topology file, each computer is characterized by 7 parameters. The first parameter, **`s`**, called *scalability* determines how many non-interacting processes may run on the computer in parallel without loss of speed. This is useful, for example, if the computer is a multiprocessor workstation. If the field `[number_of_processors]` is specified in the VPM description file, the parameter **`s`** represents the number of processors. The second parameter, **`p`**, determines the performance of the computer demonstrated on executed of some serial test code. One can see that the computer `alpha` is the most powerful and the computer `gamma` is the least powerful. Note that at runtime **`HMPI_Recon`** updates the value of the parameter for each participated computer.

The third parameter, **`n`**, determines the total number of parallel processes to run on the computer. One can see that five processes are run on each computer.

The fourth parameter determines the scalability of the communication layer provided by the computer. In this case, all computers provide serial communication layers.

Finally, the last three parameters determine the speed of point-to-point data transfer between processes running on the same computer as a function of size of the transferred data block. The first of them specifies the speed of transfer of a data block of 64 bytes (measured in bytes per second), and the second and third specify that of 64^2 and 64^3 bytes corresponding.

The homogeneous communication space of higher level is also characterized by those three parameters. Besides, the layer is detected as a parallel communication layer with factors 0.49 and 0.97 characterizing the level of parallelism of broadcast and gather correspondingly.

3.8.2 Model of HMPI program

All processes constituting the target HMPI program are divided into two groups: special process, the so-called *dispatcher*, playing the role of computing space manager, and common processes.

The dispatcher works as a server. It receives requests from the common processes and sends them commands.

At any time of the target program running, any process is either free (unemployed) or a member (employed) of one or several HMPI groups. Employing processes in created groups and dismissing them are the responsibility of the dispatcher. The only exception is the *host-process* representing the pre-defined virtual host processor, which always maps onto the first process associated with the user's terminal (host computer). Thus, just after initialization of HMPI runtime the computing space is represented by the host and a set of temporarily free (unemployed) processes. The main problem in managing processes is employing them in HMPI groups and dismissing them. The solution to this problem establishes the whole structure of a HMPI program and forms the requirements for the synchronization functions and the functions of the RTSS.

During the HMPI runtime initialization, the host-process reads and parses the topology file to initialize the HMPI runtime environment. The topology file is an ASCII file, which is generated during the creation of a virtual parallel machine on which the HMPI applications are executed. This file contains information on the model of the executing network of computers. The topology information is then sent to the dispatcher, which stores the information.

To create a HMPI group, its parent sends a creation request to the dispatcher. The request contains the full topological information on the group to be created. To compute the topological information, the parent uses the set of functions generated by the compiler from the description of the performance model of the parallel algorithm. Consider the description of the performance model **`Nbody`** shown in Figure 3.9 and the generated code shown in appendix A.

The function **`MPC_NetType_Nbody_power`** returns the number of processes in the group. Since the RTSS uses a linear numeration of processes from `0` to `n-1`, where `n` is the total number of processes, the functions **`MPC_NetType_Nbody_coord2number`** and **`MPC_NetType_Nbody_number2coord`** convert the coordinates of a process into its linear number and *vice versa*. This linear numeration is determined by the lexicographic ordering on the set of coordinates of processors. The function **`MPC_NetType_Nbody_parent`** returns the linear number of the parent process. The function **`MPC_NetType_Nbody_node`** returns the type and relative performance of the specified processor. The function **`MPC_NetType_Nbody_link`** returns the length of the directed link connecting a pair of processors. And finally the function **`MPC_NetType_Nbody_mapping`** estimates the time of execution of the parallel algorithm for a mapping.

On the other hand, the dispatcher keeps information on the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm. Based on the topological information sent by the parent and the performance model of the executing network of computers that it stores, the dispatcher selects a set of free processes, which are the most appropriate to be employed in the group to be created. After that, it sends to every free process a message saying that whether the process is employed in the group or not.

To destroy a HMPI group, its parent sends a message to the dispatcher. Note, that the parent remains employed in other groups, which share it with the group to be destroyed. In HMPI, groups are not absolutely independent of each other. Every newly created group has exactly one process shared with already existing groups. That process is the *parent* of this newly created group, and is the connecting link, through which results of computations are passed if the group ceases to exist. The rest of the members of the destroyed group become free and begin waiting for commands from the dispatcher.

Any process can detect its member/free status. It is employed and not free if a call to function **`HMPI_Is_free`** returns **`false`**. Otherwise the process is free. Any process can detect if it is a member of a HMPI group. A HMPI group is represented via its descriptor. If the descriptor **`gid`** corresponds to a HMPI group, then a process is a member of the HMPI group if and only if the function call **`HMPI_Is_member(&gid)`** returns **`true`**. In this case, the descriptor **`gid`** allows the process to obtain comprehensive information about the group as well as identify itself in the group.

Creating a group involves its parent, all free processes and the dispatcher. The parent of the group calls the function

```
HMPI_Group_create(&gid, &perf_model, modelp)
```

where **`gid`** is the group descriptor, **`perf_model`** is the handle to the performance model, and **`modelp`** are the parameters to the performance model. For the creation of the first HMPI group, the host-process can be used as the parent. The function **`HMPI_Group_create`** computes all the topological information and sends a creation request to the dispatcher. In the meantime, free processes are waiting for commands from the dispatcher at a so-called *waiting point* in the function call

```
HMPI_Group_create(&gid, &perf_model, NULL)
```

A free process leaves the waiting point either after it becomes employed in the group the descriptor of which is pointed to by **`gid`** or after the dispatcher sends, to all free processes, the command to leave the current waiting point.

3.8.3 Structure of HMPI program

The basic HMPI program model permits and supports the creation of no more than one group at a time and the existence of no more than one group at a time. The structure of a HMPI program using the basic model is explained using an example shown below:

```
/* 1 */    int main(int argc, char **argv) {
/* 2 */        void *modelp;
/* 3 */        HMPI_Group gid1, gid2;
/* 4 */        HMPI_Init(argc, argv);
/* 5 */        // Parent sends a creation request to the dispatcher.
/* 6 */        // After the group is created, Host sends a message
/* 7 */        // to the dispatcher. After receiving the message the
/* 8 */        // dispatcher sends all free processes a command ordering
/* 9 */        // them to leave the waiting point on line 14
/* 10 */       if (HMPI_Is_host())
/* 11 */          HMPI_Group_create(&gid1, &perf_model, modelp);
/* 12 */       // Free processes wait here for commands from dispatcher
/* 13 */       if (HMPI_Is_free())
/* 14 */          HMPI_Group_create(&gid1, &perf_model, NULL);
/* 18 */       // Execution of the algorithm by the members of the group
/* 19 */       if (HMPI_Is_member(&gid1)) {…}
/* 20 */       // Parent sends a destroy request to the dispatcher
/* 21 */       // Other members of the group become free here
/* 22 */       if (HMPI_Is_member(&gid1))
/* 23 */          HMPI_Group_free(&gid1);
/* 24 */       // The process is repeated for the creation of the second
/* 25 */       // group
/* 26 */       if (HMPI_Is_host())
/* 27 */          HMPI_Group_create(&gid2, &perf_model, modelp);
/* 28 */       // Free processes wait here for commands from dispatcher
/* 29 */       if (HMPI_Is_free())
/* 30 */          HMPI_Group_create(&gid2, &perf_model, NULL);
/* 31 */       // Execution of the algorithm by the members of the group
/* 32 */       if (HMPI_Is_member(&gid2)) {…}
/* 33 */       // Parent sends a destroy request to the dispatcher
/* 34 */       // Other members of the group become free here
/* 35 */       if (HMPI_Is_member(&gid2))
/* 36 */          HMPI_Group_free(&gid2);
/* 37 */       // Free processes wait here for commands from dispatcher
/* 38 */       // All common processes sync here
/* 39 */       HMPI_Finalize(0);
/* 40 */     }
```

If creation of more than one group in parallel and existence of more than one group is required, additional synchronization functions are provided, which must be used to write correct HMPI programs. However it is to be noted that no synchronization functions are necessary if the basic model is followed. We have noticed that the programming becomes complicated when synchronization functions are used. From the applications experimented with HMPI, we have observed that such complicated programming is not required. However, we explain the usage of the additional synchronization functions below for completeness.

```
/* 1 */   int main(int argc, char **argv) {
/* 2 */       void *modelp;
/* 3 */       HMPI_Group gid;
/* 4 */       HMPI_Init(argc, argv);
/* 5 */       // Parent sends a creation request to the dispatcher
/* 6 */       if (HMPI_Is_host())
/* 7 */          HMPI_Group_create(&gid, &perf_model, modelp);
/* 8 */       // Free processes wait here for commands from dispatcher
/* 9 */       if (HMPI_Is_free())
/* 10 */         HMPI_Group_create(&gid, &perf_model, NULL);
/* 11 */      // Host sends a message to the dispatcher. After receiving
/* 12 */      // the message the dispatcher sends all free processes a
/* 13 */      // command ordering them to leave the waiting point on line 10
/* 14 */      if (HMPI_Is_host())
/* 15 */         HMPI_Notify_free_processes();
/* 16 */      // Epilogue of waiting point. All common processes sync here
/* 17 */      // Execution of the algorithm by the members of the group
/* 18 */      if (HMPI_Is_member(&gid)) {…}
/* 19 */      // Parent sends a destroy request to the dispatcher
/* 20 */      // Other members of the group become free here
/* 21 */      if (HMPI_Is_member(&gid))
/* 22 */         HMPI_Group_free(&gid);
/* 23 */      // Free processes wait here for commands from dispatcher
/* 24 */      if (HMPI_Is_free())
/* 25 */         HMPI_Wait_free_processes();
/* 26 */      if (HMPI_Is_host())
/* 27 */         HMPI_Notify_free_processes();
/* 28 */      // Epilogue of waiting point. All common processes sync here
/* 29 */      HMPI_Finalize(0);
/* 30 */   }
```

In general, a HMPI program involving the creation of a HMPI group has two waiting points. The first waiting point on line 10 is called the *creation waiting point*. Here free processes wait for commands on group creation. The second waiting point on line 25 is called the *destruction waiting point*. Here the free processes wait for commands on group destruction in the function call **`HMPI_Wait_free_processes`** on line 25. In general, free processes not only participate in the creation/destruction of HMPI groups but also take part in overall computations (that is, in computations distributed over the entire computing space) and/or in the creation and destruction of HMPI groups defined in nested blocks.

The coordinated arrival of all the common processes must be made sure at the epilogue of the waiting points at lines 16 and 28. The following steps ensure it:

- It must be made sure that all other employed processes, which might send a creation/destruction request expected in the waiting point, have already reached the epilogue. This can be ensured by putting barriers using the function call **`HMPI_Barrier`**;
- The host sends a message to the dispatcher in the function call **`HMPI_Notify_free_processes`** (shown on lines 15 and 27) saying that any creation/destruction request expected at this waiting point will not come yet and that the free processes should be ordered to leave the waiting point (shown on lines 10 and 25 respectively);
- After receiving the message, the dispatcher sends all free processes a command ordering them to leave the waiting point;
- After receiving the command, each free process leaves the waiting point and reaches the epilogue.

```
int main(int argc, char **argv) {
    HMPI_Group gid1, gid2, gid3;
    HMPI_Init(argc, argv);
    if (HMPI_Is_host())
       HMPI_Group_create(&gid1, &perf_model, modelp);
    // Creation waiting point 1
    if (HMPI_Is_free())
       HMPI_Group_create(&gid1, &perf_model, NULL);
    if (HMPI_Is_host())
       HMPI_Notify_free_processes();
    // Epilogue of waiting point 1
    // Execution of the algorithm by the members of the group gid1
    if (HMPI_Is_parent(&gid1))
       HMPI_Group_create(&gid2, &perf_model, modelp);
    // Creation waiting point 2
    if (HMPI_Is_free())
       HMPI_Group_create(&gid2, &perf_model, NULL);
    if (HMPI_Is_member(&gid1))
       HMPI_Barrier(&gid1);
    if (HMPI_Is_host())
       HMPI_Notify_free_processes();
    // Epilogue of waiting point 2
    // Execution of the algorithm by the members of the group gid2
    if (HMPI_Is_member(&gid2))
       HMPI_Group_free(&gid2);
    // Destruction waiting point 3
    if (HMPI_Is_free())
       HMPI_Wait_free_processes();
    if (HMPI_Is_host())
       HMPI_Notify_free_processes();
    if (HMPI_Is_member(&gid1))
       HMPI_Barrier(&gid1);
    // Epilogue of waiting point 3
    if (HMPI_Is_parent(&gid1))
       HMPI_Group_create(&gid3, &perf_model, modelp);
    // Creation waiting point 4
    if (HMPI_Is_free())
       HMPI_Group_create(&gid3, &perf_model, NULL);
    if (HMPI_Is_member(&gid1))
       HMPI_Barrier(&gid1);
    if (HMPI_Is_host())
       HMPI_Notify_free_processes();
    // Epilogue of waiting point 4
    // Execution of the algorithm by the members of the group gid3
    if (HMPI_Is_member(&gid3))
       HMPI_Group_free(&gid3);
    // Destruction waiting point 5
    if (HMPI_Is_free())
       HMPI_Wait_free_processes();
    if (HMPI_Is_host())
       HMPI_Notify_free_processes();
    // Epilogue of waiting point 5
    if (HMPI_Is_member(&gid1))
       HMPI_Group_free(&gid1);
    // Destruction waiting point 6
    if (HMPI_Is_free())
       HMPI_Wait_free_processes();
    if (HMPI_Is_host())
       HMPI_Notify_free_processes();
    // Epilogue of waiting point 6
    HMPI_Finalize(0);
}
```

Figure 3.25: HMPI program illustrating the coordinated arrival of processes at the epilogues of the waiting points.

Figure 3.25 shows an example HMPI program demonstrating the coordinated arrival of processes at epilogues of the waiting points.

3.9 Summary

We have presented HMPI, an extension of MPI for programming high-performance computations on heterogeneous networks of computers. The main idea of HMPI is to automate the process of selection of a group of processes, which would execute the heterogeneous algorithm faster than any other group. HMPI provides features that allow the user to carefully design their parallel applications that can run efficiently on HNOCs. The features that affect the efficiency of the process of selection are:

- The accuracy of the performance model designed by the application programmers to describe their implemented heterogeneous algorithm. The performance model definition language is used to describe their implemented heterogeneous algorithm. It provides comprehensive features to express many scientific parallel applications. These features allow the application programmers to design all types of performance models ranging from the simplest to most complicated, and not very accurate to most accurate for their parallel applications.
- The accuracy of **`HMPI_Recon`**. The accuracy of **`HMPI_Recon`** depends upon how accurately the benchmark code provided by the application programmers reflects the core computations of each phase of their parallel applications. If the benchmark code provided is an accurate measurement of the core computations in each phase, **`HMPI_Recon`** gives an accurate measure of the speeds.
- The accuracy of **`HMPI_Timeof`**. The accuracy of the estimation by **`HMPI_Timeof`** is dependent upon the following:
  - The accuracy of the performance model,

- The quality of the heuristics designed for the set of parameters provided to the performance model,
- The accuracy of the model of the executing network of computers. This depends on the accuracy of the measurements of the processor speeds given by **`HMPI_Recon`** and the communication model of the executing network of computers. Currently the communication model used in HMPI runtime system is static. Future works would address the issue of efficiently updating the parameters of communication model at runtime.

From the performance models presented in this chapter and in Chapter 5 on HMPI application programming, it can be seen that a performance model can be written that is generic enough to be used for any type of data distribution. The generality of the performance model is achieved through using generic parameters in its parameter list and using data partitioning HPDI API (presented in Chapter 4) in the body of the performance model. Such performance models are only written once and used for different types of data distribution.

Thus HMPI provides all the features to the user to write portable and efficient parallel applications on HNOCs.

In the next chapter, we present Heterogeneous Data Partitioning Interface (HDPI) that automates one of the important stages of application development on HNOCs, namely, decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes. This step of heterogeneous decomposition is parameterized by the speeds of processors and the latencies and bandwidths of the communication links between them, the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy on each machine.

CHAPTER 4

The Heterogeneous Data Partitioning Interface (HDPI): Overview, Design and Preliminary Results

Parallel solution of regular and irregular problems on a HNOCs typically consists of following macro-steps:

1. Determination of characterization parameters relevant to both the computational requirements of the applications and the machine capabilities of the heterogeneous system using information about the expected types of application problems and the machines in the heterogeneous system.
2. Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes.
3. The mapping of these parallel processes to the computers of the network.
4. Application program execution on the HNOCs.

An *irregular* problem is characterized by some inherent coarse-grained or large-grained structure. This structure implies a quite deterministic decomposition of the whole problem into relatively small number of subtasks, which are of different size and can be solved in parallel. Correspondingly, a natural way of decomposition of the whole program, which solves the irregular problem on a network of computers, is a set of parallel processes, each solving its subtask and all together interacting via message passing. As sizes of these subtasks are typically different, the processes perform different volumes of computation. Therefore, the mapping of these processes to the computers of the executing HNOC should be performed very carefully to ensure the best execution time of the program.

The most natural decomposition of a *regular* problem is a large number of small identical subtasks that can be solved in parallel. As those subtasks are identical, they are all of the same size. Multiplication of two $n \times n$ dense matrices is an example of a regular problem. This problem is naturally decomposed into n^2 identical subtasks, each of which is to compute one element of the resulting matrix. The main idea behind an efficient solution to a regular problem on a heterogeneous network of computers is to transform the problem into an irregular problem, the structure of which is determined by the structure of the executing network rather than the structure of the problem itself. So, the whole regular problem is decomposed into a set of relatively large sub-problems, each made of a number of small identical subtasks stuck together. The size of each subproblem, that is, the number of elementary identical subtasks constituting the subproblem, depends on the speed of the processor, on which the subproblem will be solved. Correspondingly, the parallel program, which solves the problem on the heterogeneous network of computers, is a set of parallel processes, each solving one subproblem on a separate physical processor and all together interacting via message passing. The volume of computations performed by each of these processes should be proportional to its speed.

Thus, while step 2 of problem decomposition is trivial for irregular problems, it becomes key for a regular problem. In fact, at this very step the application programmer designs a heterogeneous data parallel algorithm by working out a generic decomposition of the regular problem parameterized by the number and speed of processors. Most typically the generic decomposition takes the form of data partitioning.

Existing programming systems for heterogeneous parallel computing [AKL+99, LAK+00, Las02] automate the steps 1, 3, and 4 of application development on HNOCs, that is, provide features that determine the characterization parameters of applications run on HNOCs, support the mapping of parallel algorithms to the executing network, and the execution of applications on HNOCs. However, they provide very poor support for generic heterogeneous decomposition of regular problems implied by the number and speed of processors. The application programmers need to solve corresponding data partitioning problems and design and implement all supportive code from scratch. Our own experience with using mpC and HMPI for parallel solution of regular problems on networks of computers has shown how tedious and error-prone this step of application development can be.

This motivated us to try and automate the step of heterogeneous decomposition of regular problems by designing a library of functions solving typical partitioning problems for networks of heterogeneous computers. Our original approach was to do it by just collecting existing algorithms, designing an API to these algorithms and implementing the API. The main problem we came across was that no classification of partitioning problems was found which could be used as a basis of API design. Existing algorithms created a very fragmented picture. Therefore the main goal of our research became to classify partitioning problems for networks of heterogeneous computers. Such classification had to help to specify problems with known efficient solutions and identify open problems. Then based on this classification an API would have to be designed and partially implemented (for problems that have known efficient solutions). An additional requirement to this classification was that it had to be useful for distributed computing on networks as well.

Our approach to classification of partitioning problems is based on two corner stones:

- A realistic performance model of networks of heterogeneous computers,
- A natural classification of mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) solving problems on networks of heterogeneous computers.

The main contributions in this chapter are:

a) The design of HDPI API based on a realistic performance model of networks of heterogeneous computers.

b) The design of efficient set partitioning algorithms using a realistic performance model of networks of heterogeneous computers. These algorithms solve the problem of optimal distribution of computational tasks on a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds cannot be accurately approximated by constant functions of problem size.

This chapter is structured as follows.

- Section 4.1 presents a realistic performance model of networks of heterogeneous computers.
    - Section 4.1.1 presents an efficient procedure for building a piecewise linear function approximation of the speed function of a processor with hierarchical memory structure. The procedure tries to minimize the experimental time used for building the speed function approximation.
- Section 4.2 presents the list of mathematical objects commonly used in parallel and distributed algorithms.
- Section 4.3 presents the classification of the problems encountered during partitioning of sets. Based on this classification, we suggest an API for partitioning sets.
- Section 4.4 presents the classification of the problems encountered during partitioning of dense matrices. Based on this classification, we suggest an API for partitioning dense matrices.
- Section 4.5 presents the classification of the problems encountered during partitioning of graphs. Based on this classification, we suggest an API for partitioning graphs.

- Section 4.6 presents the classification of the problems encountered during partitioning of trees. Based on this classification, we suggest an API for partitioning trees.
- Section 4.7 presents algorithms of partitioning sets.
    - Section 4.7.1 presents the formulation of a problem of partitioning of an **n**-element set over **p** heterogeneous processors using the performance model presented in section 4.1. We present an efficient solution to the problem of the complexity $\mathbf{O}(\mathbf{p}^2 \times \log_2 \mathbf{n})$.
    - Section 4.7.2 presents the formulation of a problem of partitioning of an **n**-element set over **p** heterogeneous processors when there is an upper bound on the size of the task that can be solved by each processor. We extend the performance model presented in section 4.1 for solving this problem and give an efficient solution to the problem of the complexity $\mathbf{O}(\mathbf{p}^3 \times \log_2 \mathbf{n})$.

4.1 A Realistic Performance Model of Networks of Heterogeneous Computers

This section presents a performance model of a network of heterogeneous computers that integrates some of the essential features of a heterogeneous network of computers having a major impact on the performance, such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging.

A number of algorithms of parallel solution of scientific and engineering problems on HNOCs have been designed and implemented [CQ93, CQ95, KL01, BBR+01]. They use different performance models of HNOCs to distribute computations amongst the processors involved in their execution. All the models use a single positive number to represent the speed of a processor, and computations are distributed amongst the processors such that their volume is proportional to this speed of the processor. Cierniak *et al.* [CLZ97] use the notion of normalized processor speed (NPS) in their machine model to solve the problem of scheduling parallel loops at compile time for HNOCs. NPS is a single number and is defined as the ratio of time taken to execute on the processor under consideration, with respect to the time taken on a base processor. In [BBP+01] and [PD99], normalized cycle-times are used, i.e. application dependent elemental computation times, which are computed via small-scale experiments (repeated several times, with an averaging of the results). Several scheduling and mapping heuristics have been proposed to map task graphs onto HNOCs [TSA+97, MS98a, IO98]. These heuristics employ a model of a heterogeneous computing environment that uses a single number for the computation time of a subtask on a machine. Yan *et al.* [YZS96] use a two-level model to study performance predictions for parallel computing on HNOCs. The model uses two parameters to capture the effects of an owner workload. These are the average execution time of the owner task on a machine and the average probability of the owner task arriving on a machine during a given time step.

However these models are efficient only if the relative speeds of the processors involved in the execution of the application are a constant function of the size of the problem and can be approximated by a single number. This is true mainly for homogeneous distributed memory systems where:

- The processors have almost the same size at each level of their memory hierarchies, and
- Each computational task assigned to a processor fits in its main memory.

But these models become inefficient in the following cases:

- The processors have significantly different memory structure with different sizes of memory at each level of memory hierarchy. Therefore, beginning from some problem size, the same task will still fit into the main memory of some processors and stop fitting into the main memory of others, causing the paging and visible degradation of the speed of these processors. This means that their relative speed will start significantly changing in favor of non-paging processors as soon as the problem size exceeds the critical value.
- Even if the processors of different architectures have almost the same size at each level of the memory hierarchy, they may employ different paging algorithms resulting in different levels of speed degradation for the task of the same size, which again means the change of their relative speed as the problem size exceeds the threshold causing the paging.

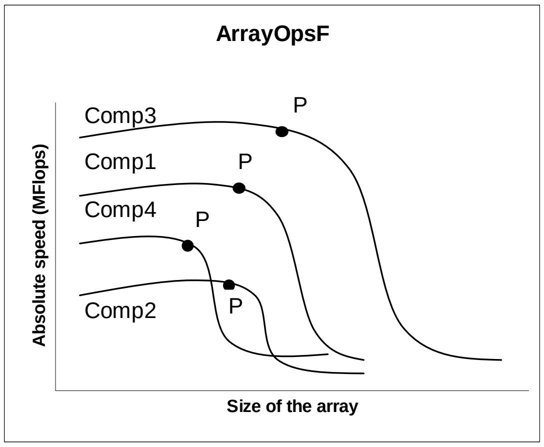
Thus considering the effects of processor heterogeneity, memory heterogeneity, and the effects of paging significantly complicates the design of algorithms distributing computations in proportion with the relative speed of heterogeneous processors. One approach to this problem is to just avoid the paging as it is normally done in the case of parallel computing on homogeneous multi-processors. However avoiding paging in local and global HNOCs may not make sense because in such networks it is likely to have one processor running in the presence of paging faster than other processors without paging. It is even more difficult to avoid paging in the case of distributed computing on global networks. There may not be a server available to solve the task of the size you need without paging.

Therefore, to achieve acceptable accuracy of distribution of computations across heterogeneous processors in the possible presence of paging, a more realistic performance model of a set of heterogeneous processors is needed. Therefore we suggest a model where the speed of each processor is represented by a continuous and relatively smooth function of the problem size whereas standard models use a single number to represent the speed. This model integrates some of the essential features underlying applications run on general-purpose common heterogeneous networks, such as the processor heterogeneity in terms of the speeds of the processors, the memory heterogeneity in terms of the number of memory levels of the memory hierarchy and the size of each level of the memory hierarchy, and the effects of paging. This model is application-centric in the sense that generally speaking different applications will characterize the speed of the processor by different functions.

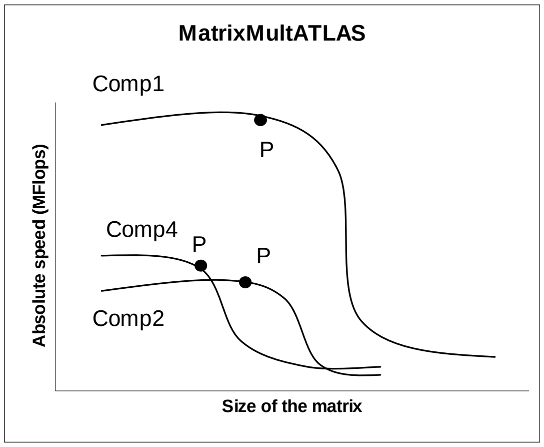
In this model, we do not incorporate one feature, which has a significant impact on the optimal distribution of computations over heterogeneous processors. This feature is the latency and the bandwidth of the communication links interconnecting the processors. This factor can be ignored if the contribution of communication operations in the total execution time of the application is negligible compared to that of computations. Otherwise, any algorithm of distribution of computations aimed at the minimization of the total execution time should take into account not only the heterogeneous processors but also the communication links whose maximal number is equal to the total number of heterogeneous processors squared. This significantly increases the space of possible solutions and increases the complexity of data partitioning algorithms. Any performance model must also take into account the contention that may be caused in the network. On a heterogeneous network of workstations using Ethernet as the interconnect, the performance will suffer if many messages are being sent at the same time. Therefore it is desirable to schedule a parallel program in such a way that only one processor sends a message at a given time. So optimal communication schedules must be obtained to reduce the overall communication time. The communication scheduling algorithms must be adaptive to variations in network performance and that derive the schedule at runtime based on current information

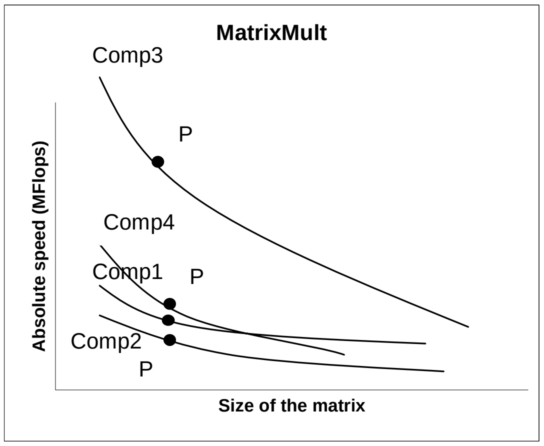
Machine Name	**Architecture**	**cpu MHz**	**Total Main Memory (kBytes)**	**Cache (kBytes)**
Comp1	Linux 2.4.20-8 Intel(R) Pentium(R) 4	2793	513304	512
Comp2	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	2048
Comp3	Windows AMD Athlon XP	3000	1030388	512
Comp4	Linux 2.4.7-10 i686	730	254524	256

Table 4.1: Specifications of four heterogeneous computers, on which applications are run to determine the effect of caching and paging in reducing their execution speed.



(a)



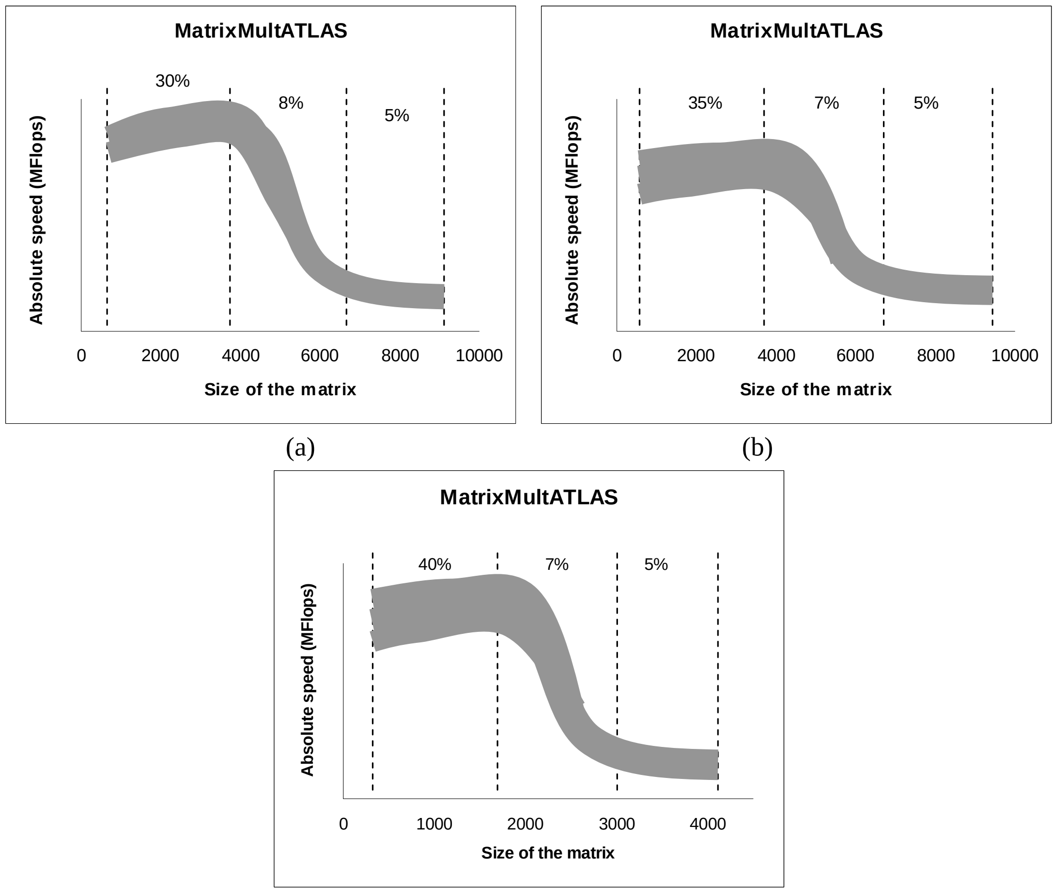


(b) (c)

Figure 4.1: The effect of caching and paging in reducing the execution speed of each of the four applications run on network of heterogeneous computers shown in Table 4.1. (a) ArrayOpsF, (b) MatrixMultATLAS, and (c) MatrixMult. P is the point where paging starts occurring.

about network load. However the problem of finding the optimal communication schedule is NP-complete. The issues involved in including the cost of communications are discussed in more detail in [DL04]. Bhat *et al*. [BPR99] present a heuristic algorithm that is based on a communication model that represents the communication performance between every processor pair using two parameters: a start-up time and a data transmission rate. The incorporation of communication cost in our functional model and subsequent derivation of efficient data partitioning algorithms using this model is a subject of our future research. In this work, we intend to fully focus on the impact of the heterogeneity of processors on optimal distribution of computations.

There are two main motivations behind the representation of the speed of the processor by a continuous and relatively smooth function of the problem size. First of all, we want the model to adequately reflect the behavior of common, not very carefully designed applications. Consider the experiments with a range of applications differently using memory hierarchy that are presented in [LT04] and shown in Figure 4.1. Carefully designed applications **ArrayOpsF** and **MatrixMultAtlas**, which efficiently use memory hierarchy, demonstrate quite a sharp and distinctive performance curve of dependence of the absolute speed on the problem size. For these applications, the speed of the processor can be approximated by a step-wise function of the problem size. At the same time, application **MatrixMult**, which implements a straightforward algorithm of multiplication of two dense square matrices and uses inefficient memory reference patterns, displays quite a smooth dependence of speed on the problem size. For such applications, the speed of the processor can not be accurately approximated by a step-wise function. It should be approximated by a continuous and relatively smooth function of the problem size if we want the performance model to be accurate enough.



(c)

Figure 4.2: Effect of workload fluctuations on the execution of application MatrixMultATLAS on computers shown in Table 4.1. The width of the performance bands is given in percentage of the maximum speed of execution of the application. (a) Performance band for Comp1, (b) Performance band for Comp2, and (c) Performance band for Comp4.

The other main motivation is that we target general-purpose common heterogeneous networks rather than dedicated high performance computer systems. A computer in such a network is persistently performing some minor routine computations and communications just as an integrated node of the network. Examples of such routine applications include email clients, browsers, text editors, audio applications, etc. As a result, the computer will experience constant and stochastic fluctuations in the workload. This changing transient load will cause a fluctuation in the speed of the computer in the sense that the execution time of the same task of the same size will vary for different runs at different times. The natural way to represent the inherent fluctuations in the speed is to use a speed band rather than a speed function. The width of the band characterizes the level of fluctuation in the performance due to changes in load over time. The shape of the band makes the dependence of the speed of the computer on the problem size less distinctive and sharp even in the case of carefully designed applications efficiently using the memory hierarchy. Therefore, even for such applications the speed of the processor can be realistically approximated by a continuous and relatively smooth function of the problem size. Figure 4.2 shows experiments conducted with application **MatrixMultATLAS** on a set of computers whose specifications are shown in Table 4.1. The application employs the level-3 BLAS routine **dgemm** [DCD+90] supplied by Automatically Tuned Linear Algebra Software (ATLAS) [WPD00]. ATLAS is a package that generates efficient code for basic linear algebra operations. The package, which contains code generators, sophisticated timers, and robust search routines, achieves this by adapting itself to differing architectures via code generation coupled with timing. The computers have varying specifications and varying levels of network integration and are representative of the range of computers typically used in networks of heterogeneous computers.

Representation of the dependence of the speed on the problem size by a single curve is reasonable for computers with moderate fluctuations in workload because in this case the width of the performance band is quite narrow. On networks with significant workload fluctuations, the speed function of the problem size should be characterized by a band of curves rather than by a single curve. In the experiments that we have conducted, we observed that computers with high level of integration into the network produce fluctuations in speed that is in the order of 40% for small problem sizes declining to approximately 6% for the maximum problem size solvable on the computer. The influence of workload fluctuations on the speed becomes less significant as the execution time increases. There is a close to linear decrease in the width of the performance band as the execution time increases. For computers with low level of integration, the width of the performance band was not greater than around 5-7% even when there was heavy file sharing activity. It is observed that for computers already engaged in heavy computational tasks, the addition of heavy loads just shifts the band to a lower level with the width of the band remaining constant, that is, the upper and lower levels of speed are reduced with the width representing the difference between the levels remaining the same. However more experimental study needs to be carried out to accurately represent the width of the performance bands for computers with varying levels of integration to increase the efficiency of the model. This is a subject of our future research where we intend to improve our functional model by adding an additional parameter that reflects the level of workload fluctuations in the network.

The functional model does not take into account the effects on the performance of the processor caused by several users running heavy computational tasks simultaneously. It supposes only one user running heavy computational tasks and multiple users performing routine computations and communications, which are not heavy like email clients, browsers, audio applications, text editors etc.

In the next section, we present a practical procedure to build a piecewise linear function approximation of the speed band of a processor, the width of the band representing the fluctuations in speed due to changes in load over time. However, the problem of efficiently building and maintaining the functional model requires further study and is open for research.

The problem of optimally scheduling divisible loads has been studied extensively and the theory is commonly referred to as Divisible Load Theory (DLT). The main features of earlier works in DLT [BGM+96, DW03a] are they assume distributed systems with a flat memory model and use a linear mathematical model where the speed of the processor is represented by a constant function of the problem size. Drozdowski and Wolniewicz [DW03b] propose a new mathematical model that relaxes the above two assumptions. They study distributed systems, which have both the hierarchical memory model and a piecewise constant dependence of the speed of the processor on the problem size. However the model they formulate is targeted mainly towards optimal distribution of arbitrary tasks for carefully designed applications on dedicated distributed multiprocessor computer systems whereas our model is aimed towards optimal distribution of arbitrary tasks for any arbitrary application on general-purpose common heterogeneous networks.

4.1.1 Procedure for Building the Functional Performance Model

We use piecewise linear function approximation illustrated in Figure 4.3 to represent the speed band of a processor, the width of the band representing the fluctuations in speed due to changes in load over time. Each of the approximations is built using a set of few experimentally obtained points. The more points used to build the approximation, the more accurate the approximation is. However it is prohibitively expensive to use large number of points. Hence an optimal set of few points needs to be chosen to build an efficient piecewise linear function approximation of the speed band. Such an approximation built gives the speed of the processor for any problem size with certain accuracy within the inherent deviation of the performance of computers typically observed in the network.

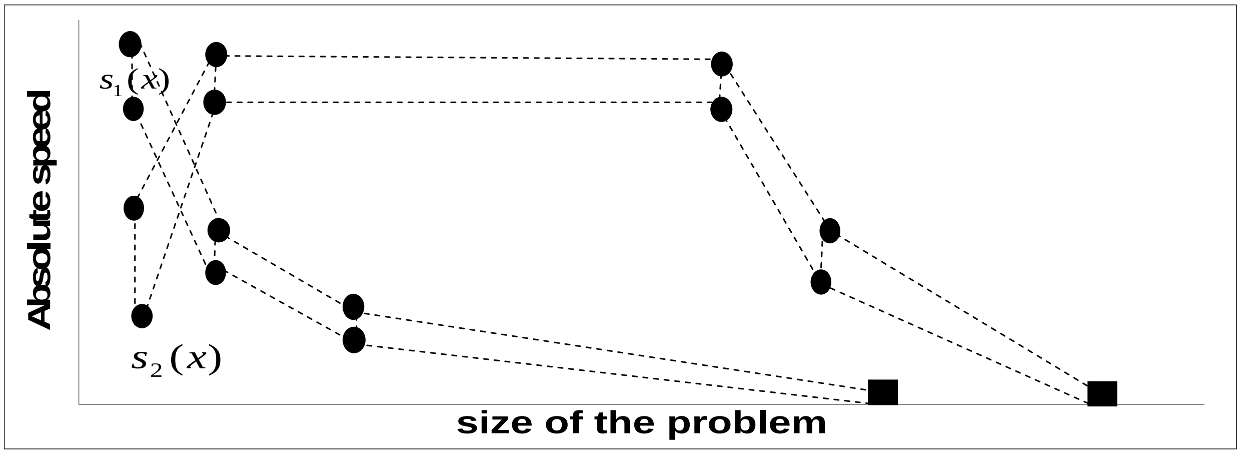


Figure 4.3: Using piecewise linear approximation to build speed bands for 2 processors. The circular points are experimentally obtained whereas the square points are calculated using heuristics. The speed band for processor $s_1(x)$ is built from 3 experimentally obtained points (application run on this processor uses memory hierarchy inefficiently) whereas the speed band $s_2(x)$ (application run on this processor uses memory hierarchy efficiently) is built from 4 experimentally obtained points.

This section is organized as follows. We start with the formulation of the speed band approximation building problem. This is followed by a section on obtaining the load functions characterizing the level of fluctuation in load over time. Then we present the assumptions adopted by our procedure and some operations and relations related to the piecewise linear function approximation of the speed band. We then explain our procedure to build the piecewise linear function approximation. And finally we demonstrate the efficiency of our procedure by performing experiments using a matrix multiplication application and a Cholesky Factorization application that use memory hierarchy efficiently and a matrix multiplication application that uses memory hierarchy inefficiently on a local network of heterogeneous computers.

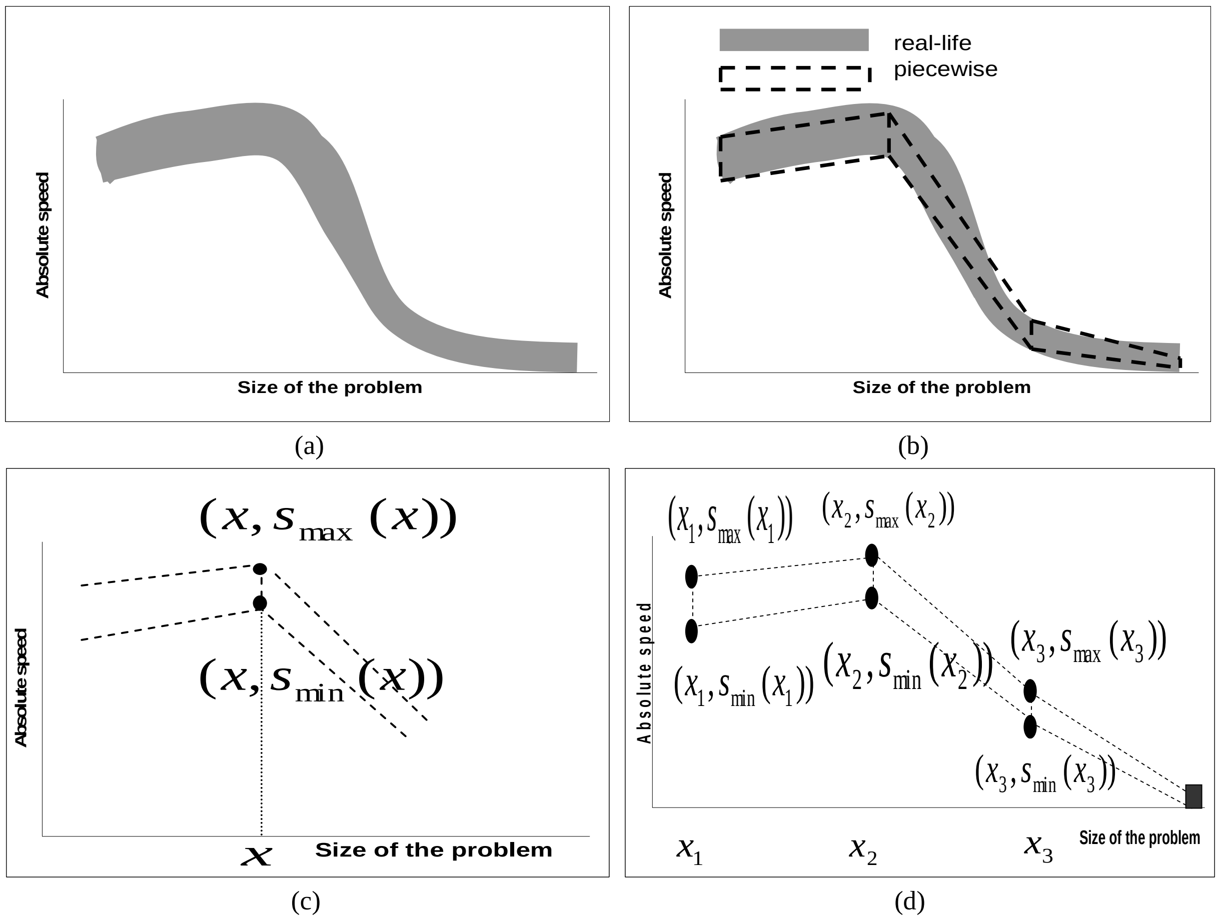


Figure 4.4: (a) Real-life speed band of a processor, (b) Real-life speed band of a processor and a piecewise linear function approximation of a processor, (c) The speeds $s_{max}(x)$ and $s_{min}(x)$ representing a cut of the real band used to build the piecewise linear approximation, and (d) Piecewise linear approximation built by connecting the cuts.

4.1.1.1 Problem Formulation

For a given application in a real-life situation, the performance demonstrated by the processor is characterized by a speed band representing the speed function of the processor with the width of the band characterizing the level of fluctuation in the speed due to changes in load over time. This is shown in Figure 4.4(a).

The problem is to find experimentally an approximation of the speed band of the processor that can represent the speed band with sufficient accuracy and at the same time spend minimum experimental time to build the approximation. One such approximation is a piecewise linear function approximation which accurately represents the real-life speed band with a finite number of points. This is shown in Figure 4.4(b).

The piecewise linear function approximation of the speed band of the processor is built using a set of experimentally obtained points for different problem sizes.To obtain an experimental point for a problem size ***x*** (we define the size of the problem to be the amount of data stored and processed by the application), we execute the application for this problem size. We measure the ideal execution time $\boldsymbol{t_{ideal}}$ and not the real time of execution. We define $\boldsymbol{t_{ideal}}$ as the time it would require to solve the problem on a completely idle processor. For example on UNIX platforms, this information can be obtained by using the ***time*** utility or the ***getrusage()*** system call. The ideal speed of execution $\boldsymbol{s_{ideal}}$ is then equal to the volume of computations divided by $\boldsymbol{t_{ideal}}$. We assume we have the load functions of historical load data $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$, which are the maximum and minimum load averages observed over increasing time periods. The load average is the number of active processes running on the processor at any time. We make a prediction of the maximum and minimum average load, $\boldsymbol{l_{max,predicted}}(\mathbf{x})$ and $\boldsymbol{l_{min,predicted}}(\mathbf{x})$ respectively, that would occur during the execution of the application for the problem size **x**. The creation of the functions $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$ and predicting the load averages are explained in detail in the next section. Using $\boldsymbol{s_{ideal}}$ and the load averages predicted, we calculate $\boldsymbol{s_{max}}(\mathrm{x})$ and $\boldsymbol{s_{min}}(\mathrm{x})$ for a problem size ***x***:

$$S_{\max}(x) = S_{ideal}(x) - l_{\min,\,predicted}(x) \times S_{ideal}(x)$$
$$S_{\min}(x) = S_{ideal}(x) - l_{\max,\,predicted}(x) \times S_{ideal}(x)$$

The experimental point is then given by a vertical line connecting the points ($\mathbf{x}$, $\mathbf{s_{max}}$(x)) and ($\mathbf{x}$, $\mathbf{s_{min}}$(x)). We call this vertical line the “cut” of the real band. This is illustrated in Figure 4.4(c). The difference between the speeds $\mathbf{s_{max}}$(x) and $\mathbf{s_{min}}$(x) represents the level of fluctuation in the speed due to changes in load during the execution of the problem size **x**. The piecewise linear approximation is obtained by connecting these experimental points as shown in Figure 4.4(d). So the problem of building the piecewise linear function approximation is to find a set of such experimental points that can represent the speed band with sufficient accuracy and at the same time spend minimum experimental time to build the piecewise linear function approximation.

Mathematically the problem of building piecewise linear function approximation can be formulated as follows:

Definition. *Piecewise Linear Function Approximation Building Problem PLFABP($l_{min}(t)$,$l_{max}(t)$):* Given the functions $\boldsymbol{l}_{\mathbf{min}}$(t) and $\boldsymbol{l}_{\mathbf{max}}$(t) ($\boldsymbol{l}_{\mathbf{min}}$(t) and $\boldsymbol{l}_{\mathbf{max}}$(t) are functions of time, characterizing the level of fluctuation in load), obtain a set of n experimental points representing the piecewise linear function approximation of the speed band of a processor, each point representing a cut given by (x_i,$s_{max}(x_i)$) and (x_i,$s_{min}(x_i)$) where x_i is the size of the problem and $s_{max}(x_i)$ and $s_{min}(x_i)$

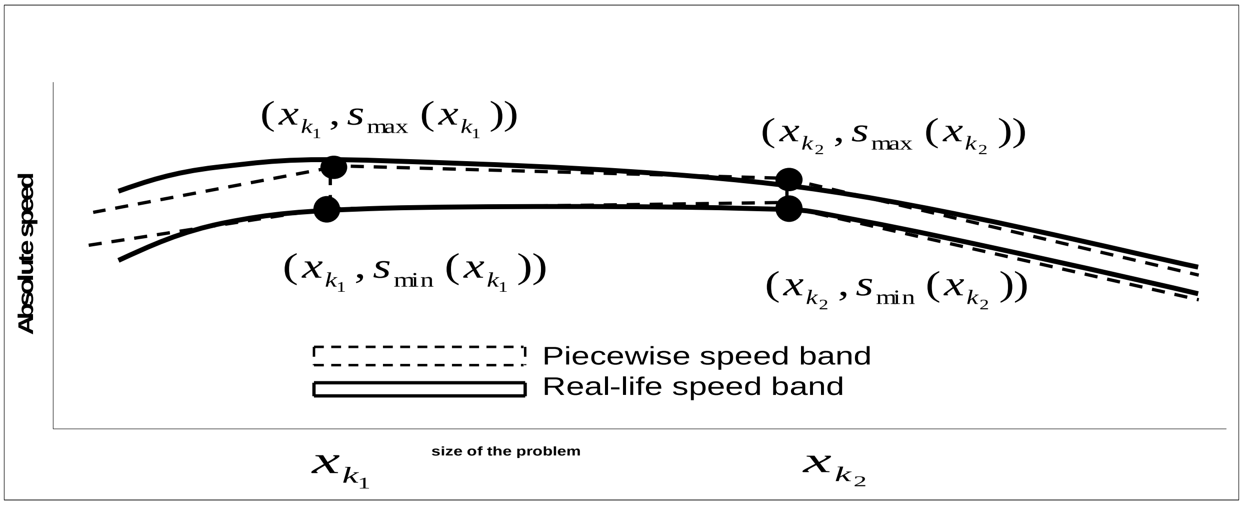

$(x_{k_1}, s_{\max}(x_{k_1}))$
$(x_{k_2}, s_{\max}(x_{k_2}))$
$(x_{k_1}, s_{\min}(x_{k_1}))$
$(x_{k_2}, s_{\min}(x_{k_2}))$
Absolute speed
Piecewise speed band
Real-life speed band
size of the problem
x_{k_1}
x_{k_2}

Figure 4.5: The non-empty intersectional area of piecewise linear function approximation with the real-life speed band is a simply connected surface.

are speeds calculated based on the functions $\boldsymbol{l}_{\mathbf{min}}$(t) and $\boldsymbol{l}_{\mathbf{max}}$(t) and ideal speed $\mathbf{s}_{\text{ideal}}$ at point **i**, such that:

- The non-empty intersectional area of piecewise linear function approximation with the real-life speed band is a simply connected surface (A surface is said to be connected if a path can be drawn from every point contained within its boundaries to every other point. A topological space is simply connected if it is path connected and it has no holes. This is illustrated in Figure 4.5), and
- the sum $\sum_{i=1}^{n} t_i$ of the times is minimal where t_i is the experimental time used to obtain point **i**.

We provide an efficient and a practical procedure to build a piecewise linear function approximation of the speed function of a processor.

4.1.1.2 Load Functions

There are a number of experimental methods that can be used to obtain the functions $\boldsymbol{l_{min}(t)}$ and $\boldsymbol{l_{max}(t)}$ (characterizing the level of fluctuation in load over time) input to our procedure for building the piecewise linear function approximation.

One of the methods is to use the metric of Load Average. Load Average measures the number of active processes at any time. High load averages usually means that the system is being used heavily and the response time is correspondingly slow. The operating system maintains three figures for averages over one, five and fifteen minute periods. There are alternative metrics available through many utilities on various platforms such as **vmstat** (UNIX), **top** (UNIX), **perfmon** (Windows) or through performance probes and they may be combined to more

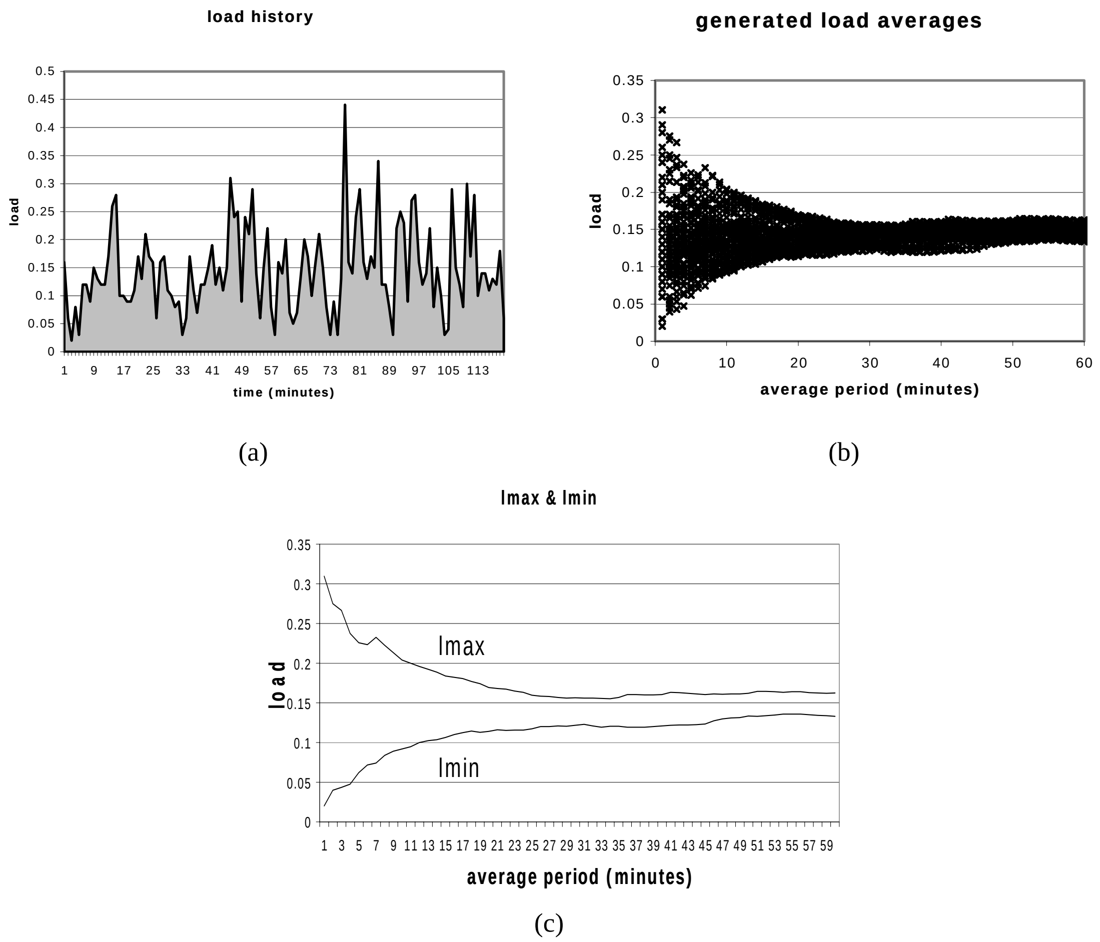


Figure 4.6: (a) $l_{max}(t)$ and $l_{min}(t)$ are generated from the load history. (b) A plot of points in matrix A. (c) $l_{max}(t)$ and $l_{min}(t)$, the maximum and minimum loads calculated from the matrix of load averages A.

accurately represent utilization of a system under a variety of conditions [WSS00]. In this work, we will use the load average metric only.

The load average data is represented by two piecewise linear functions: $\boldsymbol{l_{max}(t)}$ and $\boldsymbol{l_{min}(t)}$. The functions describe load averaged over increasing periods of time up to a limit $\boldsymbol{w}$ as shown in Figure 4.6(c). This limit should be at most the running time of the largest foreseeable problem, which is the problem size where the speed of the processor can be assumed to be zero (this is given by problem size $\boldsymbol{b}$ discussed in section on speed function approximation building procedure). For execution of a problem with a running time greater than this limit, the values of the load functions at ***w*** may be extended to infinity. The functions are built from load averages observed every Δ time units. One, five or fifteen minutes are convenient values for Δ as statistics for these time periods are provided by the operating system (using a system call ***getloadavg()***). Alternate values of Δ would require additional monitoring of the load average and translation into Δ time unit load average.

The amount of load observations used in the calculation of ***l_{max}(t)*** and ***l_{min}(t)*** is given by ***h***, the history. A sliding window with a length of ***w*** passes over the ***h*** most recent observations. At each position of the window a set of load averages is created. The set consists of load averages generated from the observations inside the window. If Δ were one minute, a one minute average would be given by the first observation in the window, a two minute average would be the average of the first and second observations in the window, and so on. While the window is positioned completely within the history, a total of ***w*** load averages would be created in each set, the load averages having periods of Δ***, 2***Δ***,*** … ***w***Δ time units. The window can move a total of ***w*** times, but after the ***(h – w)***-th time, its end will slide outside of the history. The sets of averages created at these positions will not range as far as ***w***Δ but they are still useful. From all of these sets of averages, maximum and minimum load averages for each time period Δ, ***2***Δ, … ***w***Δ are extracted and used to create the functions ***l_{max}(t)*** and ***l_{min}(t)***.

More formally, if we have a sequence of observed loads: $l_1, l_2, ..., l_h$, then the matrix ***A*** of load averages created from observations is defined as follows:

$$A = \begin{pmatrix} a_{1,1} & \cdot & \cdot & a_{1,h} \\ \cdot & & & \times \\ \cdot & & \times & \times \\ a_{w,1} & \times & \times & \times \end{pmatrix} \text{ where } a_{ij} = \frac{\sum_{k=j}^{i+j-1} l_k}{i \cdot \Delta}, \text{ for all } i = 1...h;\ j = 1...w \text{ and } \mathrm{i + j < h} \qquad (1)$$

The elements marked as × in the matrix *A* are not evaluated as the calculations would operate on observations taken beyond l_h. ***$l_{max}(t)$*** and ***$l_{min}(t)$***, are then defined by the maximum and minimum calculated ***j***-th load averages respectively, i.e. the maximum or minimum value of a row ***j*** in the matrix (see Figure 4.6). Points are connected in sequence by straight-line segments to give a continuous piecewise function. The points are given by:

$$\begin{aligned} l_{\max}(j) &= \max_{i=1}^{h}(a_{ij}) \\ l_{\min}(j) &= \min_{i=1}^{h}(a_{ij}) \end{aligned} \qquad (2)$$

Initial generation of the array has been implemented with a complexity of $h \times (w)^2$. Maintaining the functions ***$l_{max}(t)$*** and ***$l_{min}(t)$*** after a new observation is made has a complexity of w^2. Δ, ***h***, and ***w*** may be adjusted to ensure the generation and maintenance of the functions is not an intensive task.

When building the speed functions ***$S_{min}(x)$*** and ***$S_{max}(x)$***, we execute the application for a problem size ***x***. We then measure the ideal time of execution ***t_{ideal}***. We define ***t_{ideal}*** as the time it would require to solve the problem on a completely idle processor. On UNIX platforms it is possible to measure the number of CPU seconds a process has used during the total time of its execution. This information is provided by the ***time*** utility or by the ***getrusage()*** system call. We assume that the number of CPU seconds a process has used is equivalent to time it would take to complete execution on a completely idle processor: ***t_{ideal}***. We can then estimate the time of execution for the problem running under any load ***l*** with the following function:

$$t(l) = \frac{1}{1-l} \times t_{ideal} \qquad (3)$$

This formula assumes that the system is uniprocessor, that no jobs are scheduled if the load is one or greater and that the task we are scheduling is to run as a **nice**'d process (***nice*** is an

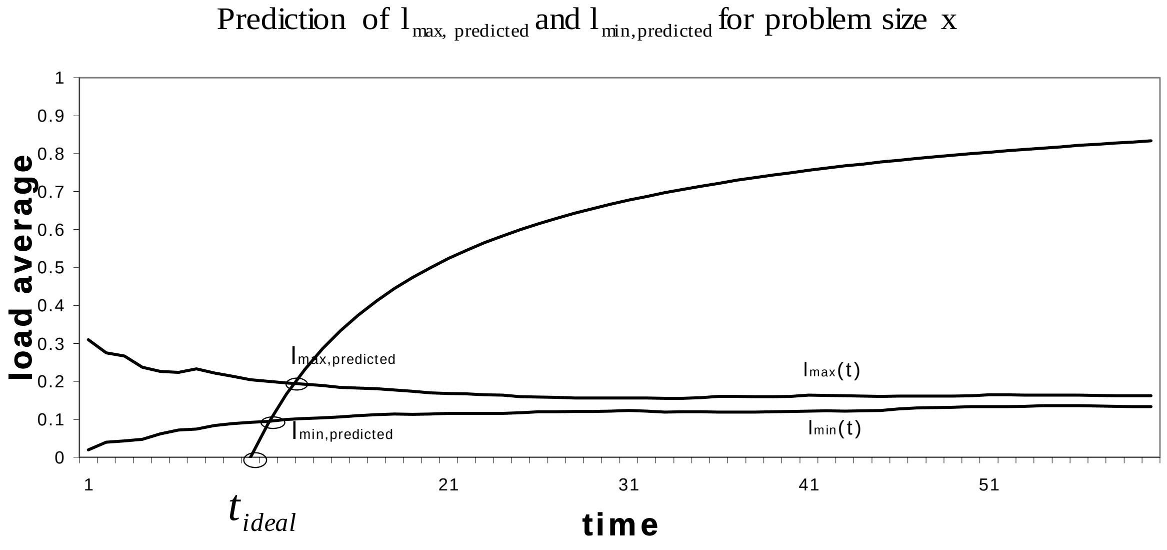


Figure 4.7: Intersection of load and running time functions (Formula 3).

operating system call that allows a process to change its priority), only using idle CPU cycles. These limitations fit the target of execution on non-dedicated platforms. If a job is introduced onto a system with a load of, for example, 0.1, the system has a 90% idle CPU, then the formula predicts that the job will take 1/0.9 times longer than the optimal time of execution: t_{ideal}.

In order to calculate the speed functions $S_{min}(x)$ and $S_{max}(x)$, we need to find the points where the function of performance degradation due to load (Formula 3) intersects with the history of maximal and minimal load $l_{max}(t)$ and $l_{min}(t)$ as shown in Figure 4.7. For a problem size **x**, the intersection points give the maximum and minimum predicted loads $l_{max,predicted}$(**x**) and $l_{min,predicted}$(**x**). Using these loads, the speeds $S_{min}(x)$ and $S_{max}(x)$ for a problem size x are calculated as:

$$S_{\max}(x) = S_{ideal}(x) - l_{\min,\,predicted}(x) \times S_{ideal}(x) \qquad (4)$$
$$S_{\min}(x) = S_{ideal}(x) - l_{\max,\,predicted}(x) \times S_{ideal}(x) \qquad (5)$$

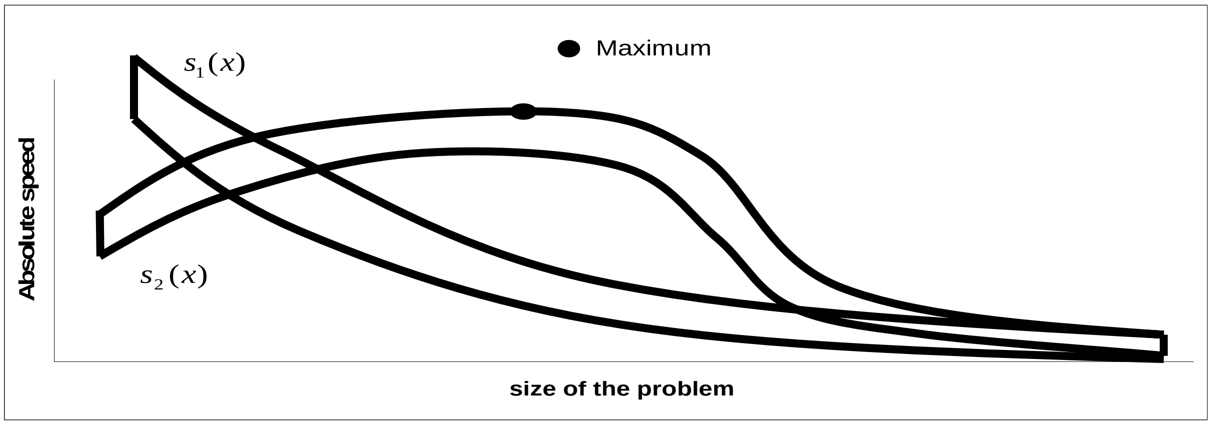
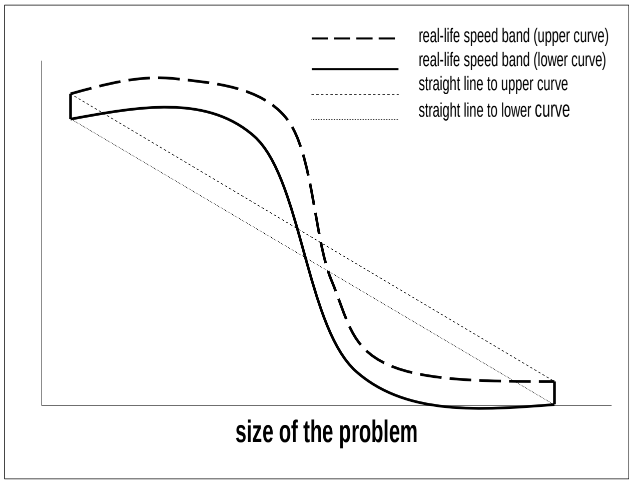
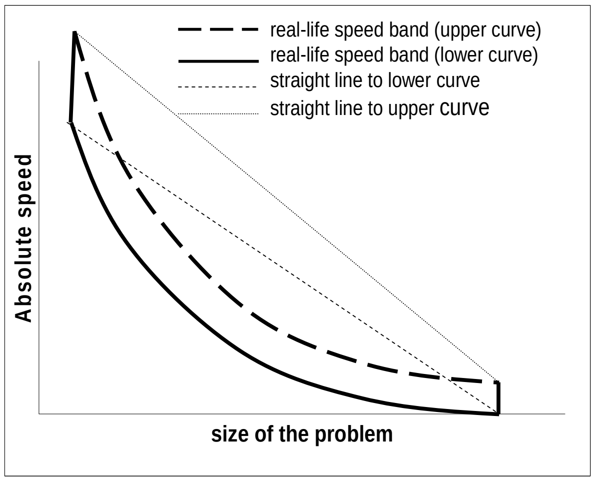


Figure 4.8: Permissible shapes of the graphs representing the real-life speed bands of two processors.



(a)



(b)

Figure 4.9: (a) Shape of real-life speed function of processor for applications that use memory hierarchy efficiently, (b) Shape of real-life speed function of processor for applications that use memory hierarchy inefficiently.

where $S_{ideal}(x)$ is equal to the volume of computations involved in solving the problem size ***x*** divided by the ideal time of execution ***t**_{ideal}*.

4.1.1.3 Assumptions

We make some assumptions on the real-life speed band of a processor. Firstly, there are some shape requirements.

(a) We assume that the upper and lower curves of the speed band are continuous functions of the size of the problem.

(b) The permissible shapes of the real-life speed band are:

- The upper curve and the lower curve are both a non-increasing function of the size of the problem for all problem sizes (as shown by $s_1(x)$ in Figure 4.8).
- The upper curve and the lower curve are both a non-decreasing function of the size of the problem followed by a non-increasing function of the size of the problem (as shown by $s_2(x)$ in Figure 4.8).

(c) A straight line intersects the upper curve of the real-life speed band in no more than one point between its endpoints and the lower curve of the real-life speed band in no more than one point between its endpoints as shown for applications that use memory hierarchy efficiently in Figure 4.9(a) and for applications that use memory hierarchy inefficiently as shown in Figure 4.9(b).

(d) We assume that the width of the real-life speed band, representing the level of fluctuations in speed due to changes in load over time, decreases as the problem size increases.

These assumptions are justified by experiments conducted with a range of applications differently using memory hierarchy presented in [LT04].

Secondly, we do not take into account the effects on the performance of the processor caused by several users running heavy computational tasks simultaneously. We suppose only one user running heavy computational tasks and multiple users performing routine computations and communications, which are not heavy like email clients, browsers, audio applications, text editors etc.

4.1.1.4 Definitions

Before we present our procedure to build a piecewise linear function approximation of the speed band of a processor, we present some operations and relations on cuts that we use to describe the procedure. The piecewise linear function approximation of the speed band of the processor is built by connecting these cuts.

1. We use I_x at problem size x to represent the interval $(s_{min}(x), s_{max}(x))$. I_x is the projection of the cut C_x connecting the points $(x, s_{min}(x))$ and $(x, s_{max}(x))$ on the y-axis.
2. $I_x \le I_y$ if and only if $s_{max}(x) \le s_{max}(y)$ and $s_{min}(x) \le s_{min}(y)$.
3. $I_x \cap I_y$ represents intersection between the intervals $(s_{min}(x), s_{max}(x))$ and $(s_{min}(y), s_{max}(y))$. If $I_x \cap I_y = \varnothing$ where Ø represents an empty set with no elements, then the intervals are disjoint. If $I_x \cap I_y = I_y$, then the interval $(s_{min}(x), s_{max}(x))$ contains the interval $(s_{min}(y), s_{max}(y))$, that is, $s_{max}(x) \ge s_{max}(y)$ and $s_{min}(x) \le s_{min}(y)$.
4. $I_x = I_y$ if and only if $I_x \le I_y$ and $I_y \le I_x$.

4.1.1.5 Speed Function Approximation Building Procedure

Procedure *Geometric Bisection Building Procedure GBBP($l_{max}(t)$,$l_{min}(t)$).* The procedure to build the piecewise linear function approximation of the speed band of a processor consists of the following steps and is illustrated in Figure 4.10:

1. We select an interval [*a*,*b*] of problem sizes where *a* is some small size and *b* is the problem size large enough to make the speed of the processor practically zero. In most cases, *a* is the problem size that can fit into the top level of memory hierarchy of the computer (L1 cache) and *b* is the problem size that is obtained based on the maximum amount of memory that can be allocated on heap. To calculate the problem size *b*, we run

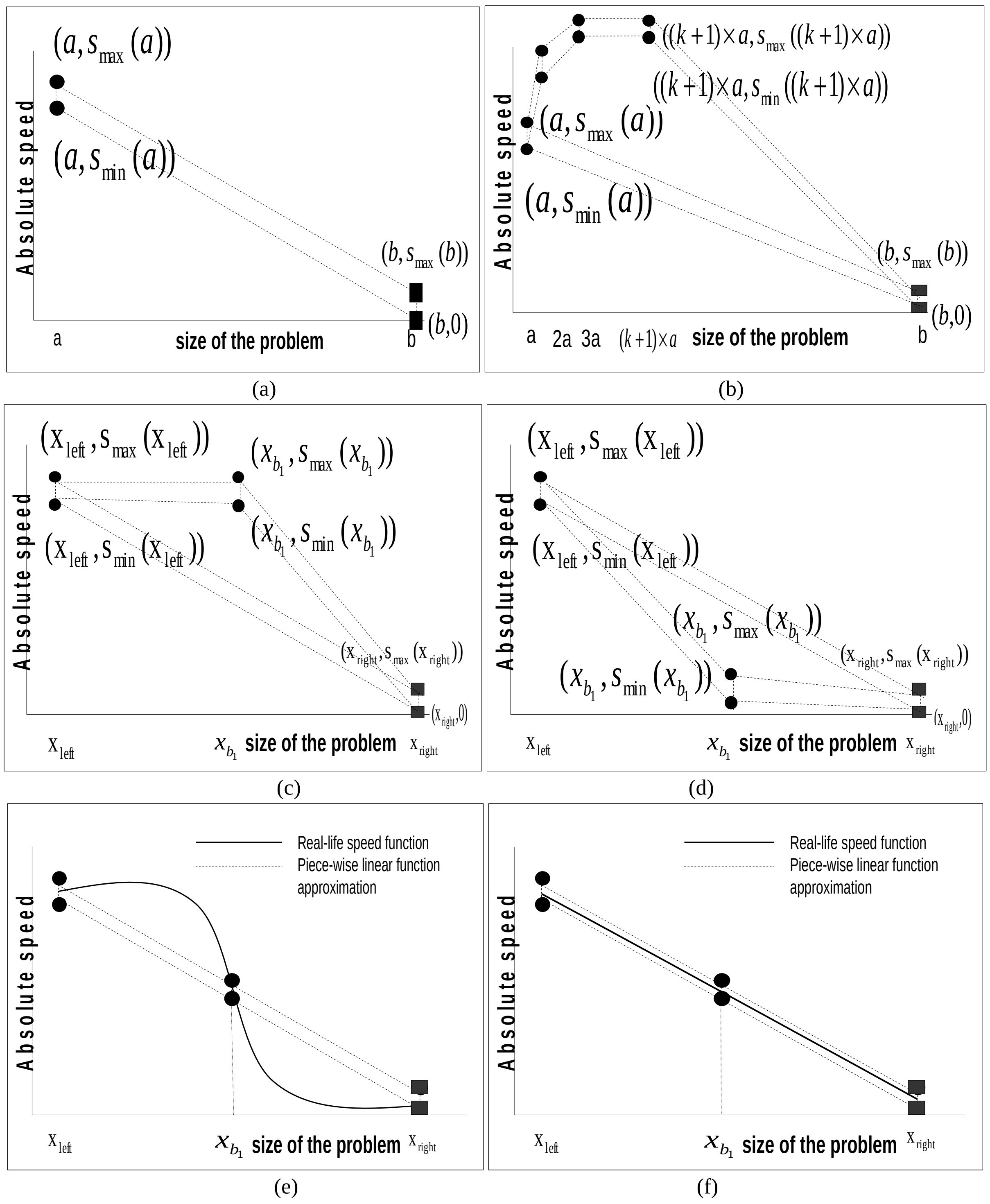

$(a, s_{max}(a))$
$(a, s_{min}(a))$
$(b, s_{max}(b))$
$(b, 0)$
Absolute speed
a
size of the problem
b
(a)
$((k+1)\times a, s_{max}((k+1)\times a))$
$((k+1)\times a, s_{min}((k+1)\times a))$
a 2a 3a $(k+1)\times a$
(b)
$(x_{left}, s_{max}(x_{left}))$
$(x_{left}, s_{min}(x_{left}))$
$(x_{b_1}, s_{max}(x_{b_1}))$
$(x_{b_1}, s_{min}(x_{b_1}))$
$(x_{right}, s_{max}(x_{right}))$
$(x_{right}, 0)$
x_{left}
x_{b_1}
x_{right}
(c)
(d)
Real-life speed function
Piece-wise linear function approximation
(e)
(f)

Figure 4.10: (a) to (f) Illustration of the procedure to obtain the piecewise linear function approximation of the speed band for a processor. Circular points are experimentally obtained points. Square points are points of intersection that are calculated but not experimentally obtained. White circular points are experimentally obtained and that fall in the current approximation of the speed band.

a modified version of the application, which includes only the code that allocates memory on heap. For example consider a matrix-matrix multiplication application of two dense square matrices *A* and *B* of size $\mathbf{n}\times\mathbf{n}$ to calculate resulting matrix *C* of size $\mathbf{n}\times\mathbf{n}$, the modified version of the application would just contain the allocation and de-allocation of matrices *A*, *B,* and *C* on heap. This modified version is then run until the application fails due to exhaustion of heap memory, the problem size at this point gives *b*. It should be noted that finding the problem size *b* by running the modified version should take just few seconds.

We obtain experimentally the speeds of the processor at point *a* given by $s_{\max}(a)$ and $s_{\min}(a)$ and we set the absolute speed of the processor at point *b* to 0. Our initial approximation of the speed band is a speed band connecting cuts C_a and C_b. This is illustrated in Figure 4.10(a).

2. We experiment with problem sizes *a* and 2*a*. If $I_{2a} \leq I_a$ or $I_{2a} \cap I_a = I_{2a}$, we replace the current approximation of the trapezoidal speed band with two trapezoidal connected bands, the first one connecting the cuts C_a and C_{2a} and the second one connecting the cuts C_{2a} and C_b. We then consider the interval [2*a*,*b*] and apply step 3 of our procedure to this interval. The speed band in this interval connecting the cuts at problem sizes 2*a* and *b* is input to step 3 of the procedure. We set x_{left} to 2*a* and x_{right} to *b*.

   If $I_a \leq I_{2a}$, we recursively apply this step until $I_{(k+1)\times a} \leq I_{ka}$ or $I_{(k+1)\times a} \leq I_{ka} = I_{(k+1)\times a}$. We replace the current approximation of the speed band in the interval [k×a,*b*] with two connected bands, the first one connecting the cuts C_{ka} and $C_{(k+1)\times a}$ and the second one connecting the cuts $C_{(k+1)\times a}$ and C_b. We then consider the interval [(k+1)×a,*b*] and apply the step 3 of our procedure to this interval. The speed band in this interval connecting the cuts $C_{(k+1)\times a}$ and C_b is input to step 3 of the procedure. We set x_{left} to (k+1)×a and x_{right} to *b*. This is illustrated in Figure 4.10(b).

It should be noted that the time taken to obtain the cuts at problem sizes {*a*, 2a, 3a,…,(k+1)×a} is relatively small (usually milliseconds to seconds) compared to that for larger problem sizes (usually minutes to hours).

3. We bisect this interval $[x_{left},x_{right}]$ into sub-intervals $[x_{left},x_{b_1}]$ and $[x_{b_1},x_{right}]$ of equal length. We obtain experimentally the cut Cx_{b_1} at problem size x_{b_1}. We also calculate the cut of intersection of the line $x=x_{b_1}$ with the current approximation of the speed band connecting the cuts Cx_{left} and Cx_{right}. The cut of intersection is given by $C'x_{b_1}$.
    a. If $Ix_{left}\cap Ix_{b_1}\neq\varnothing$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 4.10(c). We stop building the approximation of the speed band in the interval $[x_{left},x_{b_1}]$ and recursively apply step 3 for the interval $[x_{b_1},x_{right}]$. We set x_{left} to x_{b_1}.
    b. If $Ix_{left}\cap Ix_{b_1}=\varnothing$ and $Ix_{right}\cap Ix_{b_1}\neq\varnothing$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 4.10(d). We stop building the approximation of the speed band in the interval $[x_{b_1},x_{right}]$ and recursively apply step 3 for the interval $[x_{left},x_{b_1}]$. We set x_{right} to x_{b_1}.
    c. If $Ix_{left}\cap Ix_{b_1}=\varnothing$ and $Ix_{right}\cap Ix_{b_1}=\varnothing$ and $Ix_{b_1}\cap I'x_{b_1}\neq\varnothing$, then we have two scenarios illustrated in Figures 4.10(e) and 4.10(f) where experimental point at the first

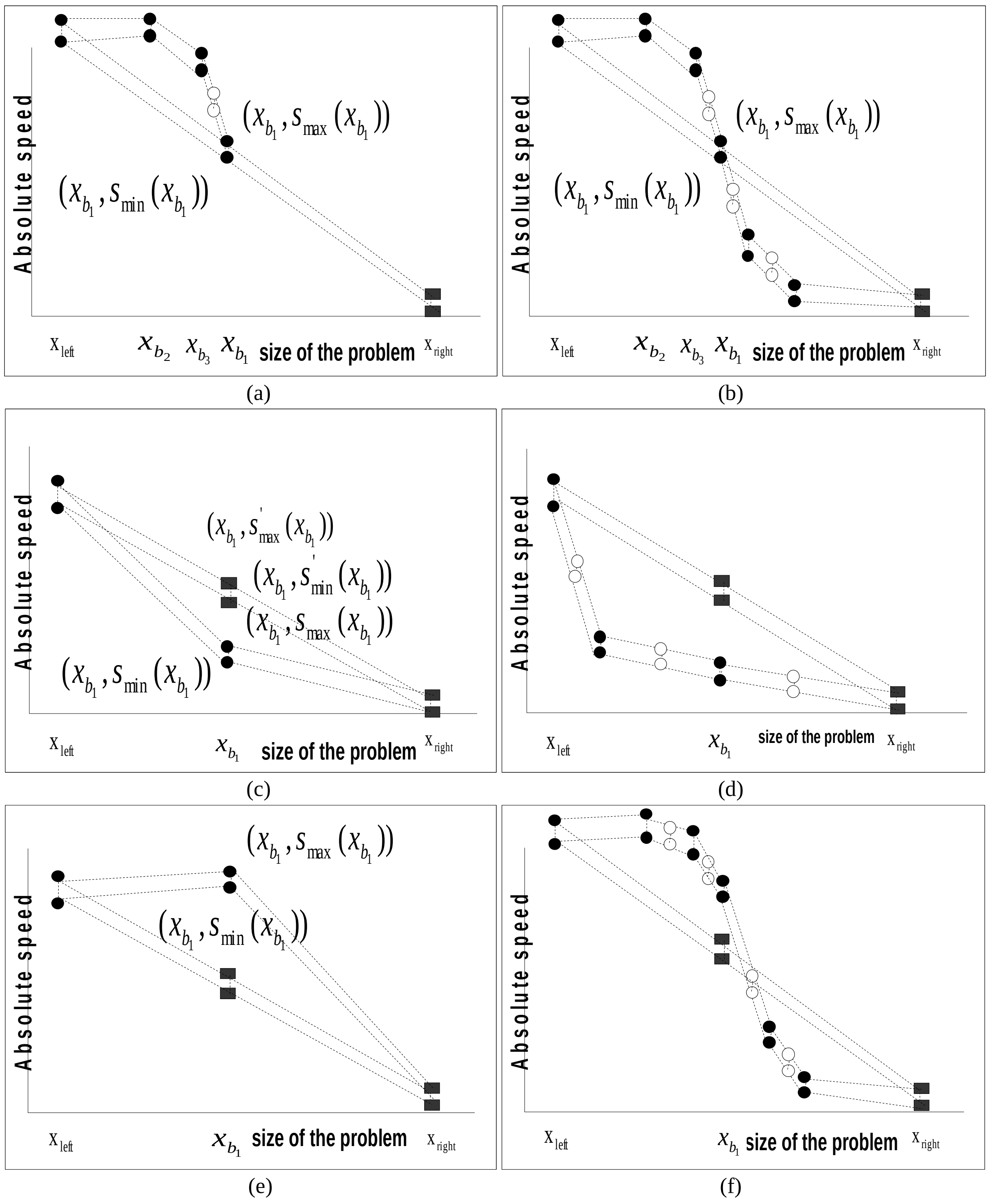

Absolute speed
size of the problem
x_{left}
x_{b_2}
x_{b_3}
x_{b_1}
x_{right}
$(x_{b_1}, s_{max}(x_{b_1}))$
$(x_{b_1}, s_{min}(x_{b_1}))$
$(x_{b_1}, s'_{max}(x_{b_1}))$
$(x_{b_1}, s'_{min}(x_{b_1}))$
(a)
(b)
(c)
(d)
(e)
(f)

Figure 4.11: (a) to (f) Illustration of the procedure to obtain the piecewise linear function approximation of the speed band for a processor. Circular points are experimentally obtained points. Square points are points of intersection that are calculated but not experimentally obtained. White circular points are experimentally obtained and that fall in the current approximation of the speed band.

point of bisection falls in the current approximation of the speed band just by accident.

Consider the interval [x_{left},x_{b_1}]. This interval is bisected at the point x_{b_2}. We obtain experimentally the cut Cx_{b_2} at problem size x_{b_2}. We also calculate the cut of intersection $C'x_{b_2}$ of the line $x=x_{b_2}$ with the current approximation of the speed band. If $Ix_{b_2} \cap I'x_{b_2} \neq \varnothing$, we stop building the approximation of the speed function in the interval [x_{left},x_{b_1}] and we replace the current approximation of the trapezoidal speed band in the interval [x_{left},x_{b_1}] with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_2} and the second one connecting the points Cx_{b_1} and Cx_{b_2}. Since we have obtained the cut at problem size x_{b_2} experimentally, we use it in our approximation. This is chosen as our final piece of our piece-wise linear function approximation in the interval [x_{left},x_{b_1}]. If $Ix_{b_2} \cap I'x_{b_2} = \varnothing$, the intervals [$x_{left}$,$x_{b_2}$] and [$x_{b_2}$,$x_{b_1}$] are recursively bisected using step 3. Figure 4.11(a) illustrates the procedure.

Consider the interval [x_{b_1},x_{right}]. This interval is recursively bisected using step 3. We set x_{left} to x_{b_1}. Figure 4.11(b) illustrates the procedure.

d. If $Ix_{left} \cap Ix_{b_1} = \varnothing$ and $Ix_{right} \cap Ix_{b_1} = \varnothing$ and $Ix_{b_1} \leq I'x_{b_1}$ and $Ix_{b_1} \cap I'x_{b_1} = \varnothing$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 4.11(c). The intervals [x_{left},x_{b_1}] and [x_{b_1},x_{right}] are recursively bisected using step 3. Figure 4.11(d) illustrates the procedure.

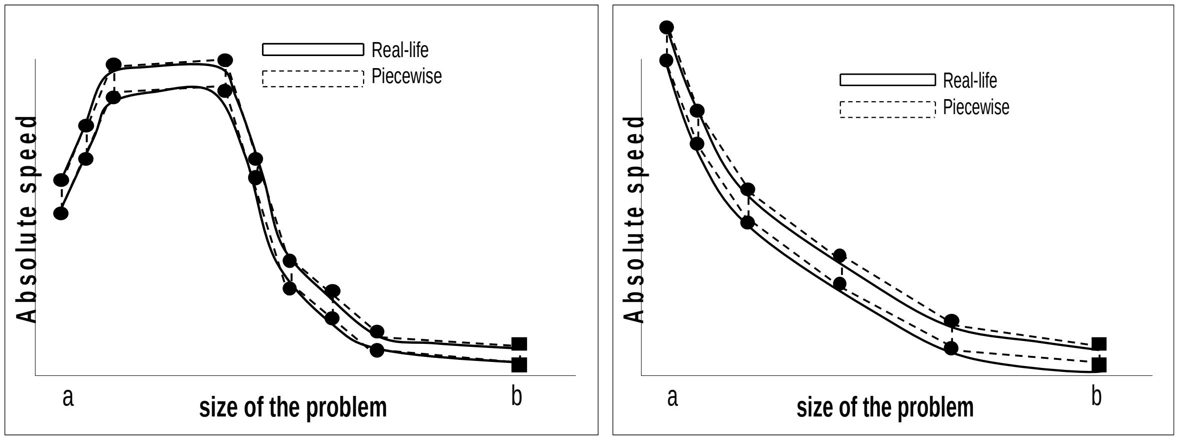


Figure 4.12: (a) The final piecewise linear function approximation of the speed band of a processor for an application that utilizes memory hierarchy efficiently. (b) The final piecewise linear function approximation of the speed band of a processor for an application that utilizes memory hierarchy inefficiently.

e. If $Ix_{left} \cap Ix_{b_1} = \emptyset$ and $Ix_{right} \cap Ix_{b_1} = \emptyset$ and $I'x_{b_1} \leq Ix_{b_1}$ and $Ix_{b_1} \cap I'x_{b_1} = \emptyset$, we replace the current approximation of the speed band with two connected bands, the first one connecting the cuts Cx_{left} and Cx_{b_1} and the second one connecting the cuts Cx_{b_1} and Cx_{right}. This is illustrated in Figure 4.11(e). The interval $[x_{left}, x_{b_1}]$ and $[x_{b_1}, x_{right}]$ are recursively bisected using step 3. Figure 4.11(f) illustrates the procedure.

4. The stopping criterion of the procedure is satisfied when we don't have any sub-interval to divide. Figures 4.12(a) and 4.12(b) show the final piecewise linear function approximation of the speed band of the processor for an application that uses memory hierarchy efficiently and an application that uses memory hierarchy inefficiently.

Processor	Architecture	cpu MHz	Total Main Memory (kBytes)	Available Main Memory (kBytes)	Cache (kBytes)	Matrix-matrix multiplication (dgemm) & Inefficient Matrix-matrix multiplication		Cholesky Factorization (dpotrf)	
						Size of matrix (n_a)	Size of matrix (n_b)	Size of matrix (n_a)	Size of matrix (n_b)
X1	Linux 2.6.8-1.521smp Intel(R) XEON(TM)	1977	1033908	460368	512	100	13000	100	19500
X2	SunOS 5.9 UltraSPARC-Iii	440	524288	401408	2048	100	7000	100	13000

Table 4.2: Specifications of two heterogeneous processors used to demonstrate the efficiency of the GBBP procedure.

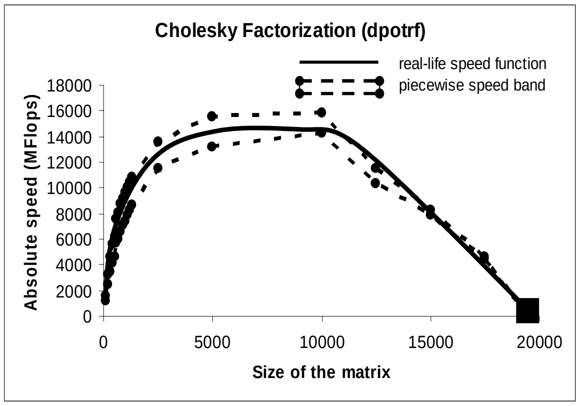
Processor	Matrix-matrix multiplication (ATLAS)	Cholesky Factorization (ATLAS)	Inefficient Matrix-matrix multiplication
	Speedup (Number of points taken to build using GBBP)	Speedup (Number of points taken to build using GBBP)	Speedup (Number of points taken to build using GBBP)
X1	8.5(7)	6.5(19)	5.9(5)
X2	5.7(10)	15(8)	5.7(5)

Table 4.3: Speedup of GBBP procedure over naïve procedure.

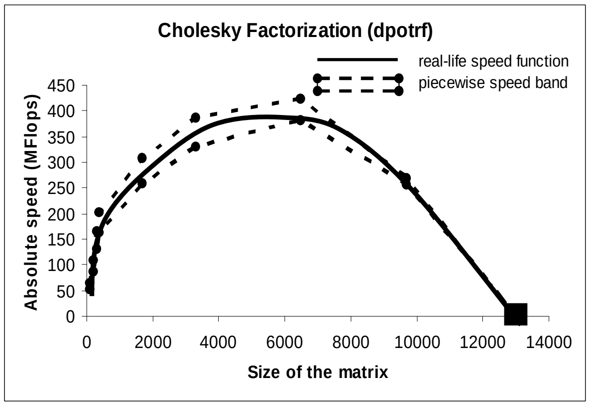
4.1.1.6 Experimental Results

We consider a Linux workstation and a Solaris workstation, which are integrated into local departmental network in the experiments. The specifications of the computer are shown in Table 4.2. The amount of memory, which is the difference between the total main memory and available main memory shown in the tables, is used by the operating system processes and few other user application processes that perform routine computations and communications such as email clients, browsers, text editors, audio applications etc. These processes use a constant percentage of CPU.

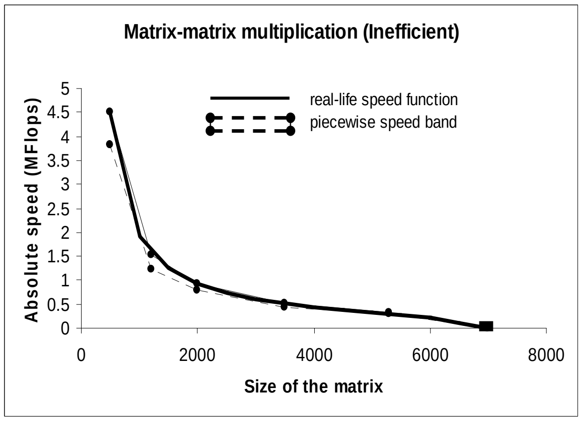
There are three applications used to demonstrate the efficiency of our procedure to build the piecewise linear function approximation of the speed band of a processor. The first application is Cholesky Factorization of a dense square matrix employing the LAPACK [ABB+92] routine **dpotrf**. The second application is matrix-matrix multiplication of two dense matrices using



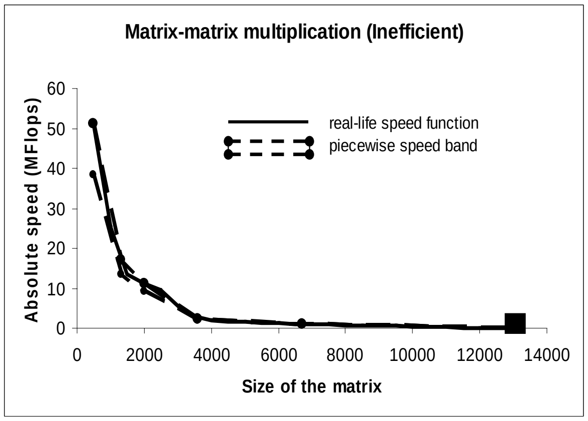
(a)



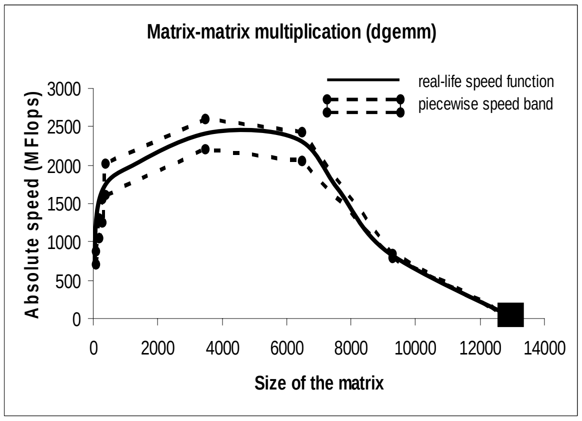
(b)



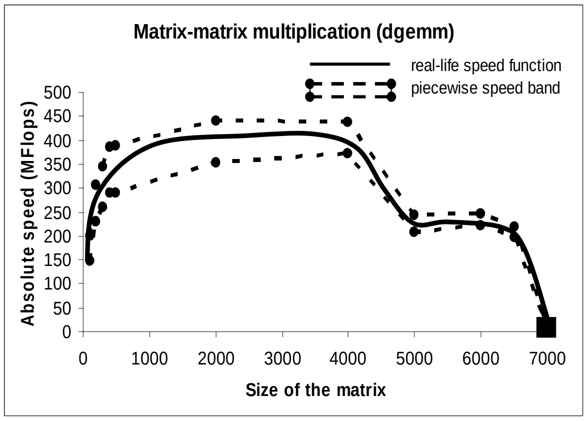
(c)



(d)



(e)



(f)

Figure 4.13: Piecewise linear approximation of the speed band against the real-life speed function. Circular points are experimentally obtained points. Square points are calculated but not experimentally obtained. (a) Cholesky Factorization using ATLAS on X1. (b) Cholesky Factorization using ATLAS on X2. (c) Matrix-matrix multiplication using memory hierarchy inefficiently on X1. (d) Matrix-matrix multiplication using memory hierarchy inefficiently on X2. (e) Matrix-matrix multiplication using ATLAS on X1. (f) Matrix-matrix multiplication using ATLAS on X2.

memory hierarchy inefficiently. The third application is based on matrix-matrix multiplication of two dense matrices employing the level-3 BLAS routine **dgemm** [DCD+90] supplied by Automatically Tuned Linear Algebra Software (ATLAS) [WPD00].

Figures 4.13(a) to (f) show the real-life speed function and the piecewise linear function approximation of the speed band of the processors X1 and X2 for the matrix multiplication and Cholesky Factorization applications. The real-life speed function for a processor is built using a set of experimentally obtained points (**x**,**s**) . To obtain an experimental point for a problem size **x**, we execute the application for the problem size at that point. The absolute speed of the processor **s** for this problem size is obtained by dividing the total volume of computations by the real execution time (and not the ideal execution time).

Table 4.3 shows the speedup of Geometric Bisection Building Procedure (GBBP) over a naïve procedure. The naïve procedure divides the interval [*a*,*b*] of problem sizes equally into **n** points. The application is executed for each of the problem sizes {(a),(a+(b-a)/n),(a+2×(b-a)/n),…,(b)} to obtain the experimental points to build the piecewise linear function approximation of the speed band. In our experiments, we have used 20 points. The speedup calculated is equal to the ratio of the experimental time taken to build the piecewise linear function approximation of the speed band using the naïve procedure over the experimental time taken to build the piecewise linear function approximation of the speed band.

We measured the accuracy of the load average functions $l_{max}(t)$ and $l_{min}(t)$ by counting how often a future load was found to be within the bounds of the curves and by measuring the area between the curves. A very wide band will encompass almost all future loads but the prediction of maximum and minimum load will be poor. We fixed ***w***, the window size, and varied ***h*** to examine how the hit ratio and area of the band changed. X1, a machine operating as a desktop

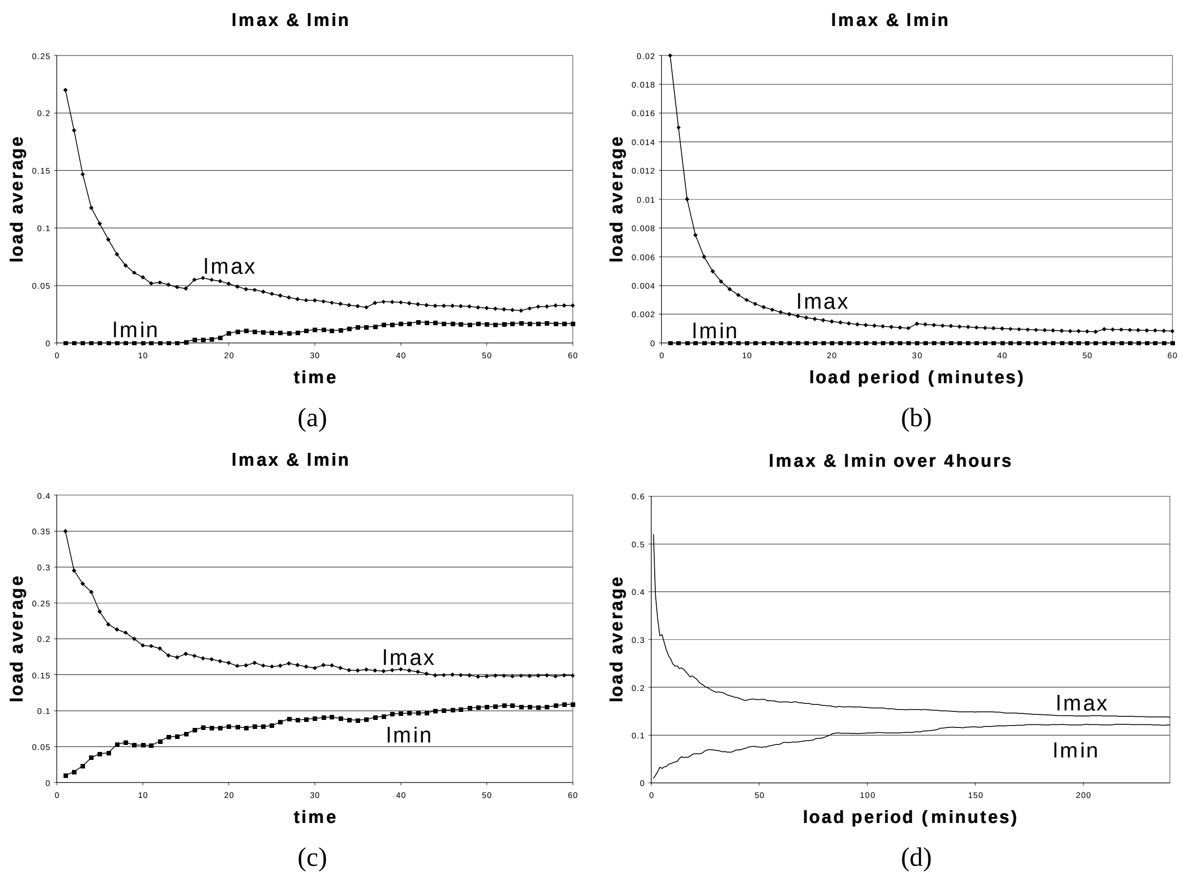


Figure 4.14: (a) Load functions for X1. (b) Load functions for X2. (c) Load functions for a departmental server running loads at all times. (d) Load functions generated with average periods beyond one hour.

with constant minor fluctuations in load, shows that a 60 minute window size gives good accuracy with 4 hours of historical data. X2 is used for running intensive jobs with relative infrequency. Figures 4.14(a) and 4.14(b) shows a sample of the load functions for processors X1-X2. Figure 4.14(c) shows a load function for a departmental server with loads running at all times.

4.1.1.7 Discussion and Future Work

Most real-life speed bands shown by applications running on variety of operating systems satisfy the requirements of the GBBP procedure. However for some operating systems, the shape of the real-life speed band has a plateau in the region of paging as shown in Figure 4.13(f), which fails the requirement (c) of the GBBP procedure. This figure shows the real-life speed function and the piecewise linear function approximation of the speed band of an UltraSparc processor X2 for a matrix multiplication application using ATLAS. In this case, due to just one plateau in the region of paging, GBBP procedure manages to build piecewise linear function approximation. However it is inefficient since it takes two additional experimental points at problem sizes 5000 and 6500 in the region 4000-7000. In general, GBBP procedure fails to build an efficient piecewise linear function approximation for such shapes. We aim to extend our procedure to build piecewise linear function approximation efficiently for such shapes.

During the building of the piecewise linear function approximation using the GBBP procedure, we consider the cut of the real-life speed band experimentally obtained for a problem size is accurate enough if there is a non-empty intersectional area with cut of the current approximation of the speed band. That is if $I_x \cap I_y \neq \Phi$ where I_x and I_y represent the intervals $(s_{min}(x), s_{max}(x))$ and $(s_{min}(y), s_{max}(y))$ of reflections of cuts C_x and C_y on y-axis respectively. The procedure thus uses implicitly the notion of distance between the intervals to represent accuracy of the building procedure. This notion of distance between the intervals can be included in the parameter list to the GBBP procedure without any modifications to the procedure.

Further consideration should be put into choosing the maximum and minimum loads to represent a particular *n* minute load average. The averages have a distribution that fits a normal curve and the limits of the load functions could be set to encompass a certain percentage of this curve. This would result in a narrower pair of load curves and could give a more accurate representation of the band.

The general shape of $l_{max}(t)$ and $l_{min}(t)$ showed that for problems executing for very long time frames, beyond one hour to one day (shown in Figure 4.14(d)), the predicted deviation in performance is less than 1.6%. Variation in load average is very small at these time scales on all our machines, despite their differing roles. This would indicate that the importance of the band lies in scheduling jobs that run for lesser periods of time. The window size, ***w***, could be dynamically assigned so that the final pair of maximum and minimum load averages represents a variation in performance of some user-defined percentage, and after this point the band could be considered a constant function.

We understand the importance of the problem of efficient maintenance of the speed function approximation of the speed band. This problem is the subject of our current research.

We aim to design efficient algorithms of data partitioning on heterogeneous networks of computers where the speed of a processor is represented by a speed band, the width of the band characterizing fluctuations in speed due to changes in load over time.

4.2 Classification of Partitioning Problems

The core of scientific, engineering or business applications is the processing of some mathematical objects that are used in modeling corresponding real-life problems. In particular, partitioning of such mathematical objects is a core of any data parallel algorithm. Our analysis of various scientific, engineering and business domains resulted in the following short list of mathematical objects commonly used in parallel and distributed algorithms: **sets** (ordered and non-ordered), **dense matrices** (and multidimensional arrangements) and **sparse matrices**, **graphs**, and **trees**.

These mathematical structures give us the second dimension for our classification of partitioning problems. In the next section, we present our approach to classification of partitioning problems using sets as mathematical objects. We also suggest an API based on the classification.

4.3 Partitioning Problems for Sets and Ordered Sets

A **set** is a well-defined collection of objects considered as a whole. The objects of a set are called elements or members. We consider the elements of the set to represent independent chunks of computations, each of equal size (i.e., each requiring the same amount of work), which can be computed without reference to each other i.e., without communication.

There are two main criteria used for partitioning a **set**:

1) The number of elements in each partition should be proportional to the speed of the processor owning that partition.
2) The sum of weights of the elements in each partition should be proportional to the speed of the processor owning that partition.

Additional restrictions that may be imposed on partitioning of an **ordered set** are:

- The elements in the set are well ordered and should be distributed into disjoint contiguous chunks of elements.

The most general problem of partitioning a **set** can be formulated as follows:

- Given: (1) A set of $\mathbf{n}$ elements with weights $\mathbf{w_i}$ ($\mathbf{i}$=0,…,$\mathbf{n}$-1), and (2) A well-ordered set of $\mathbf{p}$ processors whose speeds are functions of the size of the problem $\mathbf{x}$, $\mathbf{s_i=f_i(x)}$, with an upper bound $\mathbf{b_i}$ on the number of elements stored by each processor ($\mathbf{i}$=0,…,$\mathbf{p}$-1),
- Partition the set into $\mathbf{p}$ disjoint partitions such that: (1) The sum of weights in each partition is proportional to the speed of the processor owning that partition, and (2) The number of

elements assigned to each processor does not exceed the upper bound on the number of elements stored by it.

The most general partitioning problem for an **ordered set** can be formulated as follows:

- Given: (1) A well-ordered set of n elements with weights w_i (i=0,…,n-1), and (2) A well-ordered set of p processors whose speeds are functions of the size of the problem x, $s_i=f_i(x)$, with an upper bound b_i on the number of elements stored by each processor (i=0,…,p-1),
- Partition the set into p disjoint contiguous chunks such that: (1) The sum of weights of the elements in each partition is proportional to the speed of the processor owning that partition, and (2) The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it.

The most general partitioning problems for a **set** and an **ordered set** are very difficult and open for research. At the same time, there are a number of important special cases of these problems with known efficient solutions. The special cases are obtained by applying one or more of the following simplifying assumptions:

- All elements in the set have the same weight. This assumption eliminates n additional parameters of the problem.
- The speed of each processor is a constant function of the problem size.
- There are no limits on the maximal number of elements assigned to a processor.

One example of a special partitioning problem for a set is:

- Given: (1) A set of n elements, (2) A well-ordered set of p processors whose speeds are represented by single constant numbers, $s_0,s_1,\ldots,s_{p-1}$, and (3) There are no limits on the maximal number of elements assigned to a processor,

Model of Parallel Computation	**Weights of elements are the same**	**Weights of elements are different**
Speeds are functions of problem size & a limit exists on number of elements stored by each processor.	**Complexity** $O(p^3 \times \log_2 n)$	No known results
Speeds are functions of problem size & no limits on number of elements stored by each processor.	**Complexity** $O(p^2 \times \log_2 n)$	No known results
Speeds are single constant numbers and a limit exists on number of elements stored by each processor.	**Complexity** $O(p^3)$	NP-hard?
Speeds are single constant numbers & no limits on number of elements that each processor can hold.	**Complexity** $O(p \times \log_2 p)$	NP-hard?
Speeds are all the same (homogeneous case) & a limit exists on number of elements that each processor can hold.	**Complexity** $O(p)$	NP-hard?
Speeds are all the same (homogeneous case) & a limit exists on number of elements that each processor can hold. The sum of the limits is equal to the number of elements of the set.	**Complexity** $O(p)$	NP-hard?
Speeds are all the same (homogeneous case) & no limits on number of elements that each processor can hold.	**Complexity** $O(p)$	NP-hard?

Table 4.4: Special cases of partitioning of a set.

- Partition the set into **p** disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that partition.

The algorithm used to perform the partitioning is quite straightforward, of complexity $O(\mathbf{p}^2)$ [BBP+01]. The algorithm uses a naive implementation. The complexity can be reduced down to $O(\mathbf{p} \times \log_2 \mathbf{p})$ using ad hoc data structures [BBP+01].

Another example of a special partitioning problem for a set is:

Model of Parallel Computation	**Weights of elements are the same**	**Weights of elements are different**	
		Rearrangement of processors	
		Allowed	**Not allowed**
Speeds are functions of problem size & a limit exists on number of elements stored by each processor.	**Complexity** $O(p^3 \times \log_2 n)$	No known results	No known results
Speeds are functions of the problem size & no limits on number of elements stored by each processor.	**Complexity** $O(p^2 \times \log_2 n)$	No known results	No known results
Speeds are single constant numbers & an upper bound exists on number of elements that each processor can hold.	**Complexity** $O(p^3)$	No known results	No known results
Speeds are single constant numbers & no limits on number of elements stored by each processor.	**Complexity** $O(p \times \log_2 p)$	No known results	No known results
Speeds are all the same (homogeneous case) & a limit exists on number of elements that each processor can hold.	**Complexity** $O(p)$	No known results	No known results

Table 4.5: Special cases of partitioning of an ordered set.

- Given: (1) A set of **n** elements, (2) A well-ordered set of **p** processors whose speeds are represented by single constant numbers, $s_0, s_1, \ldots, s_{p-1}$, and (3) There is an upper bound b_i on the number of elements stored by each processor (i=0,…,**p**-1),
- Partition the set into **p** disjoint partitions such that: (1) The number of elements in each partition is proportional to the speed of the processor owning that partition, and (2) The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it

We present an algorithm to solve this problem of partitioning of complexity $O(p^3)$ in Section 4.7.2.

The special partitioning problems for a set when the speed of the processor is represented by a function of the size of the problem are:

- Given: (1) A set of **n** elements, (2) A well-ordered set of **p** processors whose speeds are functions of the size of the problem **x**, $s_i = f_i(x)$ (i=0,…,**p**-1), and (3) There are no limits on the maximal number of elements assigned to a processor,
- Partition the set into **p** disjoint partitions such that the number of elements in each partition is proportional to the speed of the processor owning that partition.

We present an algorithm of the complexity $O(p^2 \times \log_2 n)$ solving this problem is given in Section 4.7.1.

- Given: (1) A set of **n** elements, (2) A well-ordered set of **p** processors whose speeds are functions of the size of the problem **x**, $s_i = f_i(x)$ (i=0,…,**p**-1), and (3) There is an upper bound on the maximal number of elements assigned to a processor,
- Partition the set into **p** disjoint partitions such that: (1) The number of elements in each partition is proportional to the speed of the processor owning that partition, and (2) The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it

We present an algorithm of the complexity $O(p^3 \times \log_2 n)$ solving this problem is given in Section 4.7.2.

Table 4.4 and Table 4.5 summarize specific partitioning problems for a **set** and an **ordered set** respectively and their current state to the best knowledge of the authors.

Based on this classification, we suggest the following API to application programmers for partitioning a **set** and an **ordered set** respectively into **p** disjoint partitions:

```
typedef double (*User_defined_metric)(
      int p, const double *speeds, const int *actual,
      const int *ideal)

int Partition_unordered_set (
   int p, int pn, const double *speeds, const int *psizes,
   const int *mlimits, int n,  const int *w,
   int type_of_metric, User_defined_metric umf,
   double *metric, int *np)

int Partition_ordered_set (
   int p, int pn, const double *speeds, const int *psizes,
   const int *mlimits, int n,  const int *w,
   int processor_reordering, int type_of_metric,
   User_defined_metric umf, double *metric, int *np)
```

Parameter **`p`** is the number of partitions of the set. Parameters **`speeds`** and **`psizes`** specify speeds of processors for **`pn`** different problem sizes. These parameters are 1D arrays of size **`p×pn`** logically representing 2D arrays of shape **`[p][pn]`**. The speed of the **`i`**-th processor for **`j`**-th problem size is given by the **`[i][j]`**-th element of **`speeds`** with the problem size itself given by the **`[i][j]`**-th element of **`psizes`**. Parameter **`mlimits`** gives the maximum number of elements that each processor can hold.

Parameter **`n`** is the number of elements in the set, and parameter **`w`** is the weights of its elements. If **`w`** is **`NULL`**, then the set is partitioned into **`p`** disjoint partitions such that criterion (a) is satisfied. If parameters **`w`**, **`speeds`**, and **`psizes`** are all set to **`NULL`**, then the set is partitioned into **`p`** disjoint partitions such that the number of elements in each partition is the same. If **`w`** is not **`NULL`** and **`speeds`** and **`psizes`** are set to **`NULL`**, then the set is partitioned into **`p`** equally weighted disjoint partitions. If **`w`** is not **`NULL`** and **`speeds`** and **`psizes`** are not set to **`NULL`**, then the set is partitioned into **`p`** disjoint partitions such that criterion (b) is satisfied.

Parameter **`type_of_metric`** specifies which metric should be used to determine the quality of the partitioning. If **`type_of_metric`** is **`USER_SPECIFIED`**, then the user provides a metric function **`umf`**, which is used to calculate the quality of the partitioning. If **`type_of_metric`** is **`SYSTEM_DEFINED`**, the system-defined metric is used.

The output parameter **`metric`** gives the quality of the partitioning, which is the deviation of the partitioning achieved from the ideal partitioning satisfying the partitioning criteria. If the output parameter **`metric`** is set to **`NULL`**, then the calculation of metric is ignored.

If **`w`** is **`NULL`** and the set is not ordered, the output parameter **`np`** is an array of size **`p`**, where **`np[i]`** gives the number of elements assigned to the **`i`**-th partition. If the set is well ordered, the output parameter **`np`** is an array of size **`p+1`** where processor **i** gets the contiguous chunk of elements with indexes from **`np[i]`** upto and including **`np[i+1]-1`**.

If **`w`** is not **`NULL`** and the set is well ordered, then the user needs to specify if the implementations of this operation may reorder the processors before partitioning (Boolean parameter **`processor_reordering`** is used to do it). One typical reordering is to order the processors in the decreasing order of their speeds.

If **`w`** is not **`NULL`**, the set is well ordered and the processors cannot be reordered, then the output parameter **`np`** is an array of size **`p+1`**, where processor **`i`** gets the contiguous chunk of elements with indexes from **`np[i]`** upto and including **`np[i+1]-1`**.

If **`w`** is not **`NULL`**, the set is well ordered and the processors may be reordered, then **`np`** is an array of size **`2×p`**, where **`np[i]`** gives index of a processor and **`np[i+1]`** gives the size of the contiguous chunk assigned to processor given by the index **`np[i]`**.

If **`w`** is not **`NULL`** and the set is not ordered, then **`np`** is an array of size **`n`**, containing the partitions to which the elements in the set belong. Specifically, **`np[i]`** contains the partition number in which element **`i`** belongs to.

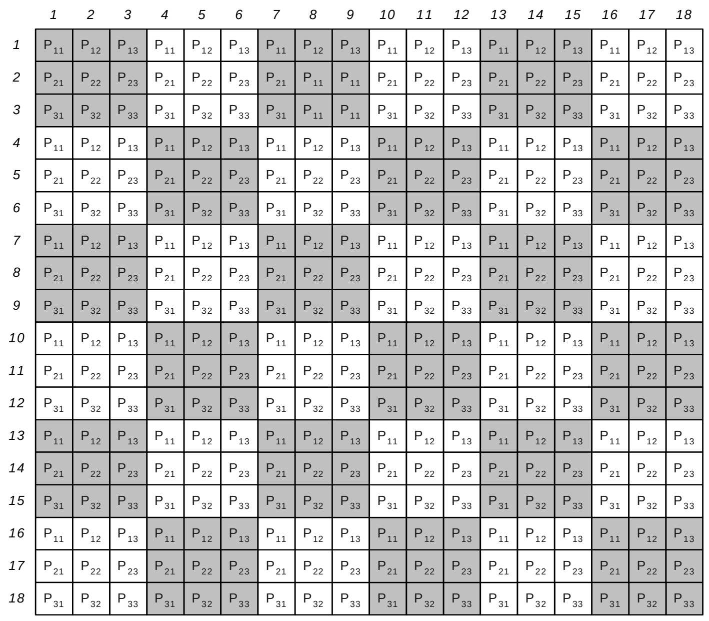
HMPI provides additional helper functions. For an **ordered set**, application programmers can use the operation, whose interface is shown below, for obtaining the coordinate of the processor owning the set element at index **`i`**.

```
int Get_set_processor (
    int i, int n, int p, int processor_reordering,
    const int *np)
```

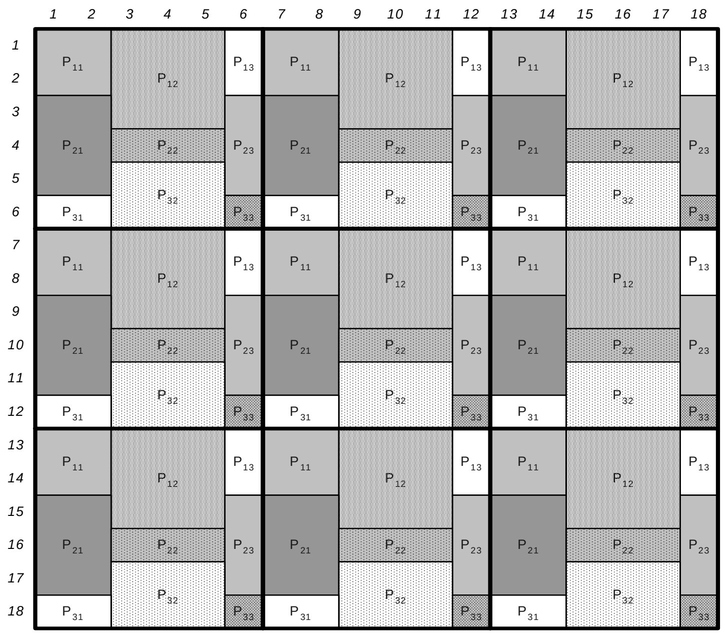
For an unordered set, application programmers can use the operation, whose interface is shown below, to obtain the number of elements allocated to processor **`i`**.

```
int Get_my_partition (
    int i, int p, const double *speeds, int n)
```

Some of the typical examples where the partitioning interfaces for sets can be used are striped partitioning of a matrix and simple partitioning of a graph. In striped partitioning of a matrix, a matrix is divided into groups of complete rows or complete columns, the number of rows or columns being proportional to speeds of the processors. In simple partitioning of an unweighted graph, the set of vertices are partitioned into disjoint partitions such that the criterion (a) is satisfied. In simple partitioning of a weighted graph, the set of vertices are partitioned into disjoint partitions such that criterion (b) is satisfied.



(a)



(b)

Figure 4.15: (a) Homogeneous two-dimensional block-cyclic distribution of a matrix with 18×18 elements over a 3×3 processor grid; (b) Heterogeneous two-dimensional block-cyclic distribution of a matrix of size 18×18 elements with 6×6 generalized blocks distributed over a 3×3 processor grid. Each labeled (shaded and unshaded) area represents different rectangles of blocks, and the label indicates at which location in the processor grid the rectangle is stored – all rectangles labeled with the same name are stored in the same processor. Each square in a bold frame represents different generalised blocks.

4.4 Partitioning Problems for Dense Matrices

A **matrix** of size $r \times c$ is a rectangular array of numbers arranged in r horizontal rows and c vertical columns.

The typical partitioning of a matrix distributes a matrix into one-dimensional or two-dimensional distributions of numbers. Actually, a matrix is a special case of a multidimensional arrangement of numbers. In general, a multidimensional distribution of numbers arranged in nd dimensions is partitioned into distributions of numbers arranged in 1or 2 … or $nd-1$ dimensions.

The typical partitioning of a matrix uses block-cyclic distribution of matrices on either a one-dimensional or a two-dimensional grid of processors. The advantages of block-cyclic distribution are easily understood. Blocked versions of the parallel algorithms for matrix multiplication and linear system solvers are used in ScaLAPACK [CDD+96] to squeeze the most out of state-of-the-art processors with pipelined arithmetic units and multilevel memory hierarchy. The block-cyclic data layout has been selected for the dense algorithms implemented in ScaLAPACK principally because of its scalability [DVW94], load balance, and communication [HW94] properties. The block cyclic distribution has also been incorporated in the HPF language [HPF94, HPF97]. Suppose we have a set of processes considered as a logical process grid with p rows and q columns and a block-partitioned matrix with block size $m \times n$. The generalized homogeneous two-dimensional block cyclic distribution partitions the matrix into *generalized blocks* of size $(m \times p) \times (n \times q)$, each partitioned into $(p \times q)$ blocks of the same size $(m \times n)$, going to separate process. In the case of generalized *heterogeneous* block-cyclic distribution, the blocks are not of the same size, but their sizes $(m_{ij} \times n_{ij})$ depend on the performance of the processors. The generalized blocks are identically partitioned into $p \times q$ unequal rectangles, each being assigned to a different process. The area of each rectangle should be proportional to the speed of the processor that stores the rectangle. Figure 4.15(a) shows homogeneous block-cyclic distribution of a matrix of size 18×18 over a 3×3 processor grid and Figure 4.15(b) shows heterogeneous two-dimensional block cyclic distribution of a matrix of size 18×18 elements with 6×6 generalized blocks over a 3×3 process grid. Note that the elements of the matrix are usually small rectangular blocks and most commonly square blocks of size **s×s**, where optimal values of **s** depend on the memory hierarchy and on the communication-to-computation ratio of the target computer. The interfaces provided in this section for partitioning a matrix are applicable even if an element of the matrix is a rectangular block of numbers or just a number.

In the figures presented in this section, the types of distributions should not be read as 1D data distributions and 2D data distributions. Instead they should be read as data distributions on a linear array of processors (1D processor arrangements) and data distributions on 2D processor grid arrangements.

The types of distribution of a matrix over a linear processor array are:

1) Horizontal or Vertical slices (shown in Figures 4.16(a) and 4.16(b) respectively),
2) Naïve row or column contiguous placement of sub-blocks (shown in Figure 4.16(c)), and
3) General rectangular distribution where the partition assigned to each processor is a rectangle (shown in Figures 4.16(e) to 4.16(f)).

For distributions (1) and (2), partitioning interfaces of sets can be used.

The general rectangular distribution is characterized by

- Optimization of some additional parameter such as minimization of the sum of half-perimeters of rectangles etc.
- Restrictions on the shape.

- Row-based.
- Column-based.

Figures 4.16(d) and 4.16(e) show generalized row-based and column-based rectangular distributions of a generalized block over a linear array of eight processors. Figure 4.16(f) shows a general rectangular distribution of a generalized block over a linear array of eight processors.

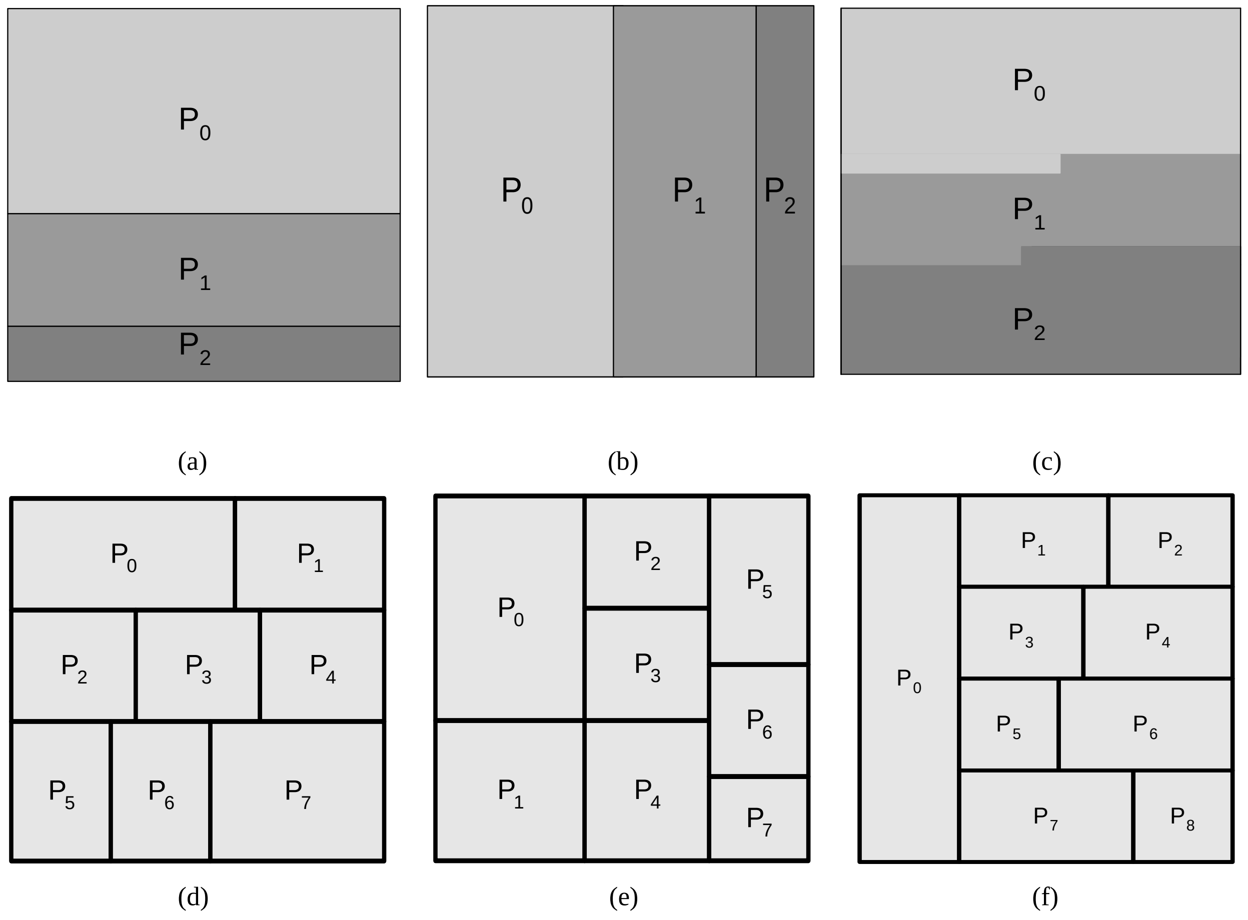


Figure 4.16: Types of distribution of a matrix over a linear processor array. (a) Horizontal sliced distribution of a generalized block over a linear array of 3 processors, (b) Vertical sliced distribution of a generalized block over a linear array of 3 processors, (c) Row-contiguous distribution of the elements of a generalized block over a linear array of 3 processors, (d) Generalized row-based distribution of a generalized block over a linear array of 8 processors, (e) Generalized column-based distribution of a generalized block over a linear array of 8 processors, and (f) General rectangular distribution of a generalized block over 9 processors.

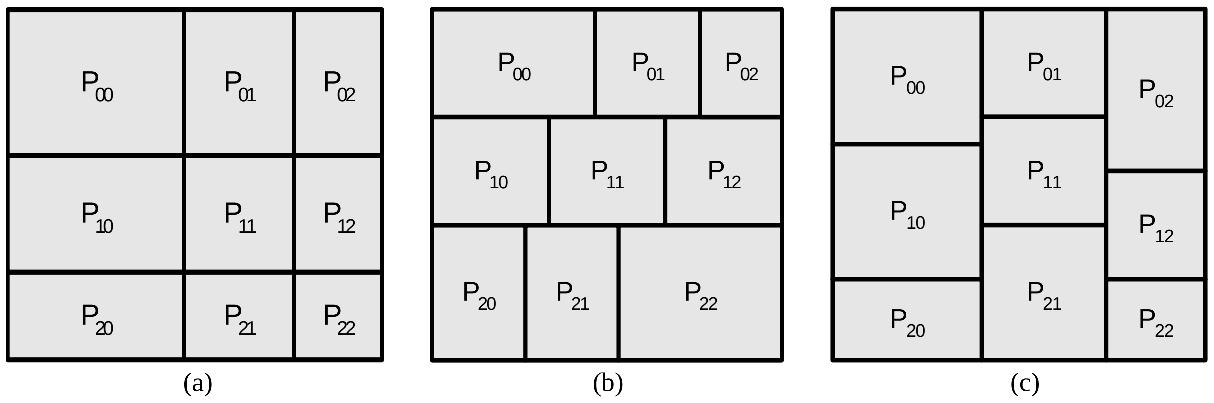


Figure 4.17: Types of distribution of a matrix over a two-dimensional grid of processors. (a) Cartesian distribution of a generalized block over a 3×3 processor grid, (b) Row-based distribution of a generalized block over a 3×3 processor grid, and (c) Column-based distribution of a generalized block over a 3×3 processor grid.

The row-based and column-based distributions over a linear processor array as shown in Figures 4.16(d) and 4.16(e) respectively are based on [BBR+01]. They present a column-based partitioning scheme where rectangles are tiled (the area of the rectangle assigned to a processor is proportional to the speed of the processor) in columns. The partitioning algorithm tries to minimize the communication volume, which is defined as the sum of the half-perimeters of the rectangles. The general rectangular distribution shown in Figure 4.16(f) is based on [CQ93, KRW96]. [CQ93] use an orthogonal recursive bisection to perform the matrix decomposition. [KRW96] devise four matrix partitioning algorithms over a linear array of processors that try to minimize the communication volume while simultaneously balancing the computational load among the processors.

The data distributions over a linear array of processors presented here solve an optimization problem, which is to minimize the communication volume while simultaneously load-balancing the computations. Hence they have wide applicability. They could be used, for example, in an application employing a finite-difference scheme where the heterogeneous processors

Model of Parallel Computation	**Linear array of processors**			**2D processor grid**	
	General Row based/Column based Rectangular		**General Rectangular**	**Cartesian**	**Row based/Column based**
Speeds are functions of the size of the problem and no limits exist on number of elements that each processor can hold.	No known results		No known results	No known results	No known results
Speeds are single constant numbers and an upper bound exists on number of elements that each processor can hold.	No known results		No known results	No known results	No known results
Speeds are single constant numbers and no limits exist on number of elements that each processor can hold.	**Complexity**		**Complexity**	**Complexity**	**Complexity**
	$F = \sum$	$F = \max$			
	Polynomial. [BBR+01]	NP-complete. [BBL+00]	NP-complete. [BBR+00]	Approximate O(1).	O(1). [KL01]

$\sum$ = Communications between the processors cannot be performed in parallel
max = Communications between the processors performed in parallel

Table 4.6: Special cases of partitioning of a dense matrix.

communicate boundary elements at each step. The communication scheme in such an application could be anything instead of just top-bottom, left-right, top-bottom-left-right, or nearest neighbor. These distributions solve exactly the same optimization problem that is associated with such communication schemes.

The most general problem of partitioning a matrix over a linear processor array can be formulated as follows: Given a matrix of size **r×c** with restrictions on the shape of partitions and a functional **F** computing some numerical characteristic of each partition and given an ordered set of **p** processors whose speeds are functions of the size of the problem $s_0 = f_0(x), s_1 = f_1(x), ..., s_{p-1} = f_{p-1}(x)$, with an upper bound **b_i** $(i = 0,1,\cdots, p-1)$ on the number of elements stored by each processor, partition the matrix into **p** disjoint rectangles such that

- The area of each rectangle is proportional to the speed of the processor owning it.
- The number of elements of the matrix assigned to each processor does not exceed the upper bound on the number of elements that can be stored by it.
- The partition minimizes (maximizes) the functional **F**.
- The partition satisfies the shape restrictions.

This is an open problem for research. Table 4.6 lists the specific cases of this problem.

The shape restrictions for data distributions over a linear processor array are: for row-based distribution as shown in Figure 4.16(d), in each row, the number of left neighbors and the number of right neighbors for each processor must be the same. For column-based distribution as shown in Figure 4.16(e), in each column, the number of top neighbors and the number of bottom neighbors for each processor must be the same.

The types of distribution of a matrix over a two-dimensional grid of processors are:

1) Cartesian,
2) Column-based, and
3) Row-based.

Figure 4.17(a) shows the Cartesian distribution of a generalized block over a 3×3 processor grid. Figure 4.17(b) shows the row-based distribution of a generalized block over a 3×3 processor grid. Figure 4.17(c) shows the column-based distribution of a generalized block over a 3×3 processor grid.

The most general problem of partitioning a matrix over a 2D processor grid is the same as the most general problem of partitioning a matrix over a linear processor array except that the matrix is partitioned into **p**×**q** rectangles where **p** is the number of processors along the row of the grid and **q** is the number of processes along the column of the grid.

This is an open problem for research. Table 4.6 lists the specific cases of this problem.

The shape restrictions for data distributions on 2D processor arrangements are

- If there is a **p**×**q** grid of processors where **p** is the number of processors along the row of the grid and **q** is the number of processes along the column of the grid, the number of rectangles in each row must be **p** and in each column must be **q**. For example if **p**=3, **q**=3, the number of rectangles in the each row is 3 and the number of rectangles in each column is 3. Also the processors are arranged in a row-major order with their row and column coordinates as follows: (0,0), (0,1), (0,2) in first row, (1,0), (1,1), (1,2) in second row, and (2,0), (2,1), (2,2) in third row. If **p**=3, **q** =4, the number of rectangles in the each row is 4 and the number of rectangles in each column is 3. The processors are arranged in a row-major order with their row and column coordinates as follows: (0,0), (0,1), (0,2), (0,3) in first row, (1,0), (1,1), (1,2), (1,3) in second row, (2,0), (2,1), (2,2), (2,3) in third row, and (3,0), (3,1), (3,2), (3,3) in fourth row.
- For Cartesian distribution as shown in Figure 4.17(a), in each row, the number of left neighbors and the number of right neighbors for each processor must be the same and in each column, the number of top neighbors and the number of bottom neighbors for each processor must be the same. For row-based distribution as shown in Figure 4.17(b), in each row, the number of left neighbors and the number of right neighbors for each processor must be the same. For column-based distribution as shown in Figure 4.17(c), in each

column, the number of top neighbors and the number of bottom neighbors for each processor must be the same.

Based on this classification, we suggest an API that allows the application programmers to partition a generalized block into **p** disjoint rectangles in the case of 1D processor arrangements and **p×q** disjoint rectangles in the case of 2D processor grid arrangements. The first operation, whose interface is shown below, is used for 2D processor grid arrangements.

```
int Partition_matrix_2d (
    int p, int q, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
   int type_of_distribution, int *w, int *h,
   int *trow, int *tcol, int *ci, int *cj )
```

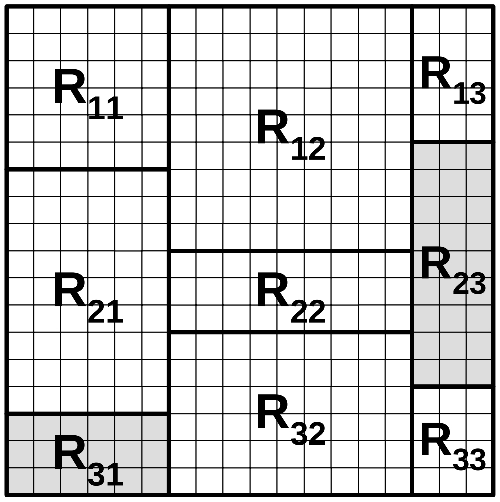
The parameter **p** is the number of processors along the row of the processor grid. The parameter **q** is the number of processors along the column of the processor grid.

Parameters **speeds** and **psizes** specify speeds of processors for **pn** different problem sizes. These parameters are 1D arrays of size **p×q×pn** logically representing arrays of shape **[p][q][pn]**. The speed of the **(i,j)**-th processor for **k**-th problem size is given by the **[i][j][k]**-th element of **speeds** with the problem size itself given by the **[i][j][k]**-th element of **psizes**. Parameter **mlimits** gives the maximum number of elements that each processor can hold.

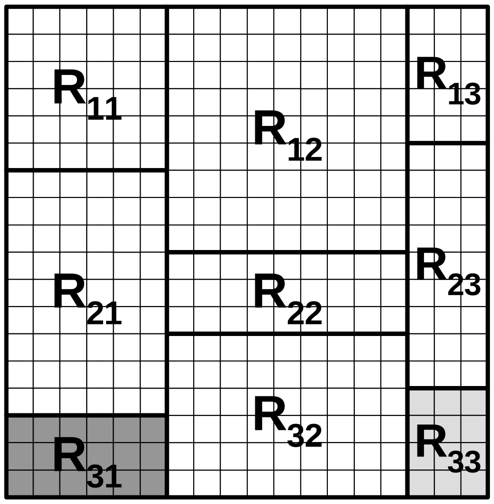
The parameters **m** and **n** are the sizes of the generalized block along the row and the column.

The input parameter **type_of_distribution** specifies if the distribution is **CARTESIAN**, **ROW-BASED**, and **COLUMN-BASED**.

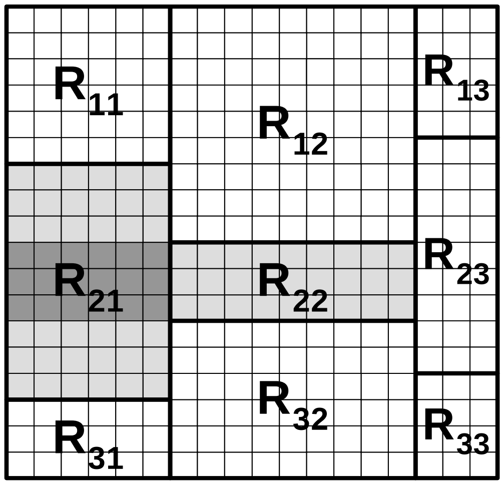
Output parameter **w** gives the widths of the rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**×**q**.



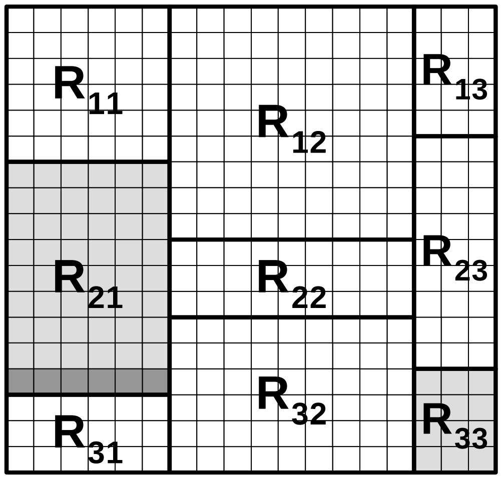
(a) R_{31} has no horizontal neighbours of R_{23}



(b) All $s \times s$ blocks of R_{31} are horizontal neighbours of R_{33}



(c) In R_{21}, horizontal neighbours of R_{22} constitute a 3×6 rectangle subarea



(d) Last row of R_{21} consists of horizontal neighbours of R_{33}

Figure 4.18: Different combinations of rectangles in a generalized block. (a) No **s×s** block of rectangle R_{31} is a horizontal neighbor of rectangle R_{23}; therefore, h[3][1][2][3] = 0. (b) All **s×s** blocks of rectangle R_{31} are horizontal neighbors of rectangle R_{33}; h[3][1][3][3] = 3. (c) Neighbors of rectangle R_{22} in rectangle R_{21} make up a 3×6 rectangle area (shaded dark grey); h[2][1][2][2] = 3. (d) Neighbors of rectangle R_{33} in rectangle R_{21} make up the last row of this rectangle (shaded dark grey); h[2][1][3][3] = 1.

Output parameter **h** gives the heights of rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**×**q**×**p**×**q** logically representing array of shape **[p][q][p][q]**. Parameter h specifies the heights of rectangle areas of a generalized block of matrix *A*, which are horizontally communicated between different pairs of abstract processors. Let R_{IJ} and R_{KL} be the rectangles of a generalized block of matrix *A* assigned to processors P_{IJ} and P_{KL} respectively. Then, **h[I][J][K][L]** gives the height of the rectangle area of R_{IJ}, which is required by processor P_{KL} to perform its computations. All heights are measured in **s×s** blocks. Figure 4.18 illustrates possible combinations of rectangles R_{IJ} and R_{KL} in a generalised block. Let us call an **s×s** block of R_{IJ} a *horizontal neighbour* of R_{KL} if the row of **s×s** blocks that contains this **s×s** block will also contain an **s×s** block of R_{KL}. Then, the rectangle area of R_{IJ}, which is required by processor P_{KL} to perform its computations, comprises all horizontal neighbours of R_{KL}. The macro **H(p, q, I, J, K, L)** gives the height **h[I][J][K][L]**.

Figure 4.18(a) shows the situation when rectangles R_{IJ} and R_{KL} have no horizontal neighbours. Correspondingly, **h[I][J][K][L]** will be zero. Figure 4.18(b) shows the situation when all **s×s** blocks of R_{IJ} are horizontal neighbours of R_{KL}. In that case, both **h[I][J][K][L]** will be equal to the height of R_{IJ}. Figures 4.18(c) and 4.18(d) shows the situation when only some of **s×s** blocks of R_{IJ} are horizontal neighbours of R_{KL}. In this case, **h[I][J][K][L]** will be equal to the height of the rectangle subarea of R_{IJ} comprising the horizontal neighbours of R_{KL}. Note that **h[I][J][K][L]** specifies the height of R_{IJ}, and **h[I][J][K][L]** will be always equal to **h[I][J][K][L]**.

Output parameter **trow** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first row of the generalized block. This parameter is an array of size **p**×**q**.

Output parameter **tcol** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first column of the generalized block. This parameter is an array of size **p**×**q**.

Output parameters **ci**, and **cj** are each an array of size **m**×**n**. The coordinates of the processor in its processor grid to which the matrix element at row **i** and column **j** of the generalized block is assigned is given by **ci[i×n+j]**, and **cj[i×n+j]** respectively. If these parameters are set to **NULL**, then they are not evaluated.

The second set of operations, whose interfaces are shown below, is used for distribution of a matrix over a 1D processor grid. These set of operations allow the application programmers to formulate heuristic solutions for their optimization problems.

```
typedef double (*Get_lower_bound)(
    int p, const double *speeds, int m, int n);
typedef double (*DP_function)(
    int rowsorcolumns, int rectangles, int p,
    const double *speeds, const double **previous_values,
    int *r);
typedef double (*Iterative_function)(
    int p, const int *w, const int *h,
    const int *trow, const int *tcol);
typedef double (*Refining_function)(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    int *w, int *h, int *trow, int *tcol);
int Partition_matrix_1d_dp(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, DP_function dpf,
    int type_of_distribution,
    int *w, int *h, int *trow, int *tcol, int *c)
int Partition_matrix_1d_iterative(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, Iterative_function cf,
    int *w, int *h, int *trow, int *tcol, int *c)
int Partition_matrix_1d_refining(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, Refining_function impf,
    int *w, int *h, int *trow, int *tcol, int *c)
```

The parameter **p** is the number of number of disjoint rectangles the matrix is partitioned into. Parameters **speeds** and **psizes** specify speeds of processors for **pn** different problem sizes. These parameters are 1D arrays of size **p×pn** logically representing 2D arrays of shape **[p][pn]**. The speed of the **i**-th processor for **j**-th problem size is given by the **[i][j]**-th element of **speeds** with the problem size itself given by the **[i][j]**-th element of **psizes**. Parameter **mlimits** gives the maximum number of elements that each processor can hold.

The parameters **m** and **n** are the sizes of the generalized block along the row and the column.

Output parameter **w** gives the widths of the rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**. Output parameter **h** gives the heights of rectangles of the generalized block assigned to different processors. This parameter is an array of size **p**×**p**. Output parameter **trow** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first row of the generalized block. This parameter is an array of size **p**. Output parameter **tcol** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first column of the generalized block. This parameter is an array of size **p**.

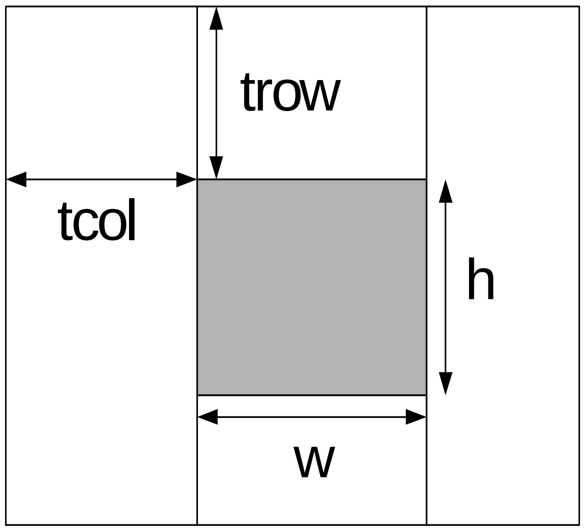


Figure 4.19: The parameter **`w`** gives the width of the rectangle of the generalized block assigned to different processors. The parameter **`h`** gives the height of rectangle of the generalized block assigned to different processors. The parameter **`trow`** gives the top leftmost point of the rectangle of the generalized block assigned to different processors from the first row of the generalized block. The parameter **`tcol`** gives the top leftmost point of the rectangle of the generalized block assigned to different processors from the first column of the generalized block.

Output parameter **`c`** is an array of size **`m`**×**`n`**. The coordinates of the processor in its processor array to which the matrix element at row **`i`** and column **`j`** of the generalized block is assigned is given by **`c[i×n+j]`**. If this parameter is set to **`NULL`**, then the parameter is ignored.

The meaning of these parameters is shown in the Figure 4.19.

For general rectangular distribution over a linear array of processors, there may be a number of optimization problems used to partition a matrix some of which are:

- Given **`p`** processors with different speeds, how to allocate data so that the length of the largest communication is optimized. In terms of tiling, how to tile the unit square into nonoverlapping rectangles of prescribed area $s_0, s_1, \cdots, s_{p-1}$ whose sum is 1 so that the largest perimeter is minimized.

- Given an array **`m×n`** of nonnegative numbers and a positive integer **`p`**, find a partition of the array into **`p`** nonoverlapping rectangular arrays such that the maximum weight of any rectangle in the partition is minimized (the weight of the rectangle is the sum of its elements).

For the operation **`Partition_matrix_1d_dp`**, the input parameter **`type_of_distribution`** specifies if the distribution is **`ROW-BASED`** or **`COLUMN-BASED`**. This operation allows the application programmers to formulate their optimization problem based on the dynamic programming paradigm. Dynamic programming views a problem as a set of interdependent subproblems. It solves subproblems and uses the results to solve larger subproblems until the entire problem is solved. The solution to the subproblem is expressed as a function of solutions to one or more subproblems at preceding levels. Application programmers provide a composition function **`dpf`**, whose nature depends on the problem. The function **`dpf`** is iteratively built by incrementing the value of parameters **`rowsorcolumns`** from `0` to **`p`** and **`rectangles`** from `0` to **`p`**. The aim is to find the optimal number of rows or columns and fitting `0` to **`p`** rectangular areas in each of these rows or columns such that the objective of the optimization problem is satisfied. Consider a step of the iteration where number of columns is 2 and the number of rectangles to fit is 3, then the following arrangements are tried: one rectangle in the first column, two rectangles in the second column and two rectangles in the first column, one rectangle in the second column. The arrangement that results in the minimum value of **`dpf`** is returned in the output parameter **`r`**. The value of this parameter gives the total number of rectangles in preceding columns, that is, the solution to the subproblem at preceding level. This option is mainly used for generalized row-based and column-based partitioning of a matrix shown in Figure 4.16(d) and Figure 4.16(e) respectively. One of the examples where this option can be used for obtaining an optimal column-based partitioning of the matrix is the column-based heuristic approach proposed by Beaumont et al [BBR01].

The operations **`Partition_matrix_1d_iterative`** and **`Partition_matrix_1d_refining`** must be used when the type of distribution is general rectangular as shown in Figure 4.16(f).

In the case of operation **`Partition_matrix_1d_iterative`**, the application programmers are allowed to provide a cost function **`cf`** that tests the optimality of a partition from a finite set of partitions. The initial partition in this finite set of partitions is obtained using a problem-specific strategy. The cost function **`cf`** is called iteratively for each of the partitions in the subset of partitions. The return value of this function gives an optimality value. At each step of the iteration, the optimality value is compared to the lower bound of the optimal solution to the optimization problem. Application programmers specify a function **`lb`**, which is used to calculate the lower bound of their optimization problem. The iteration stops when the function returns an optimality value less than or equal to the lower bound or a negative return value indicating that the partitioning cannot be improved and that the current partition is optimal.

In the case of operation **`Partition_matrix_1d_refining`**, the application programmers are allowed to provide a refinement function **`rf`** that refines an old partition giving a new better partition. A negative return value of this function suggests that the old partition cannot be refined further. This function is iteratively called. The partition for the first call of this refining function is obtained using a problem-specific strategy. Application programmers specify a function **`lb`**, which is used to calculate the lower bound of their optimization problem. The iteration stops when the refinement function **`rf`** returns an optimality value less than or equal to the lower bound indicating that the current partition is optimal.

Application programmers can use a mix of these operations to obtain an optimal partitioning of the matrix. For example, application programmers can call the operation **`Partition_matrix_1d_dp`** to obtain an initial partition, which can be input to the operation **`Partition_matrix_1d_refining`** for further refinement.

HMPI provides additional helper functions.

- **`Get_matrix_processor`** to obtain the coordinates (**i**,**j**) of the processor owning the matrix element at row **r** and column **c**.
- **`Get_my_width`** to obtain the width of the rectangle owned by the processor with coordinates (**i**,**j**).
- **`Get_my_height`** to obtain the height of the rectangle owned by the processor with coordinates (**i**,**j**).

The following helper functions are useful for dense matrix factorizations on HNOCs such as LU factorization, QR decomposition, and Cholesky factorization.

- **`Get_diagonal`** to obtain the number of the diagonal elements owned by the processor with coordinates (**i**,**j**).
- **`Get_my_elements`** to obtain the number of elements owned by the processor with coordinates (**i**,**j**) in the upper or lower half of the matrix including the diagonal elements.
- **`Get_my_kk_elements`** to obtain the number of elements owned by the processor with coordinates (**i**,**j**) in the upper or lower half of the matrix starting from (**k**,**k**) including the diagonal elements.

HMPI also provides interfaces for partitioning multidimensional arrangements of numbers.

4.5 Partitioning Problems for Graphs

A **graph** denoted ***G*** = (***V***, ***E***), consists of a nonempty set ***V*** of **vertices** (or **nodes**) and a set ***E*** of ***edges*** (or ***arcs***) such that each edge corresponds to a unique ordered pair of distinct vertices {*u*, *v*} and no more than one edge corresponds to {*u*, *v*}. The sets ***V*** and ***E*** are assumed to be finite.

There are five main criteria used for partitioning a graph:

a) The number of vertices in each partition should be proportional to the speed of the processor owning that partition.

b) The sum of weights of the vertices in each partition should be proportional to the speed of the processor owning that partition.

c) Set of disjoint partitions satisfying criterion (a) and the edgecut should be minimal. Edgecut is defined as the total weight of the edges in the graph whose incident vertices belong to different partition.

d) Set of disjoint partitions satisfying criterion (b) and the edgecut should be minimal.

The partitioning operations on sets can be used to partition a graph such that either of criterion (a) or (b) is satisfied.

The most general problem of partitioning a graph can be formulated as follows: Given a graph **G** consisting of **n** vertices $\{0,1,\cdots,n-1\}$ with weights **v**$\mathbf{v_i}$ $(i=0,1,\cdots,n-1)$ and **m** edges $\{0,1,\cdots,m-1\}$ with weights $\mathbf{e_i}$ $(i=0,1,\cdots,m-1)$ and given a linear array of **p** processors whose speeds are functions of the size of the problem $s_0=f_0(x), s_1=f_1(x),\ldots,s_{p-1}=f_{p-1}(x)$ and there is an upper bound $\mathbf{b_i}$ $(i=0,1,\cdots,p-1)$ on the number of vertices that each processor can hold, partition the graph into **p** disjoint partitions such that

- The sum of weights of the vertices in each partition is proportional to the speed of the processor owning that partition.

- The number of elements assigned to each processor does not exceed the upper bound on the number of elements stored by it.
- The edgecut is minimal.

This is an open problem for research. At the same time, there are a number of important special cases of these problems with known efficient solutions. The special cases are obtained by applying one or more of the following simplifying assumptions:

- All vertices in the graph have the same weight. This assumption eliminates **n** additional parameters of the problem.
- All edges in the graph have the same weight. This assumption eliminates **m** additional parameters of the problem.
- The speed of each processor is a constant function of the problem size.
- There are no limits on the maximal number of vertices assigned to a processor.

One example of a special partitioning problem for a graph is:

- Given: (1) A graph **G** of **n** vertices and **m** edges $\{0,1,\cdots,m-1\}$ with weights **e_i** $(i=0,1,\cdots,m-1)$, and (2) A well-ordered set of **p** processors whose speeds are represented by single constant numbers, $s_0, s_1, ..., s_{p-1}$,
- Partition the graph into **p** disjoint partitions such that: (1) The number of vertices in each partition is proportional to the speed of the processor owning that partition, and (2) The edgecut is minimal.

This is an open problem for research. Kumar, Das and Biswas [KDB02] employ a multilevel heterogeneous partitioner, called *MiniMax*, developed for distributed heterogeneous systems that differs from existing partitioners in that it allows full heterogeneity in both the system and workload characteristics. In their model, the heterogeneous system consists of processors with

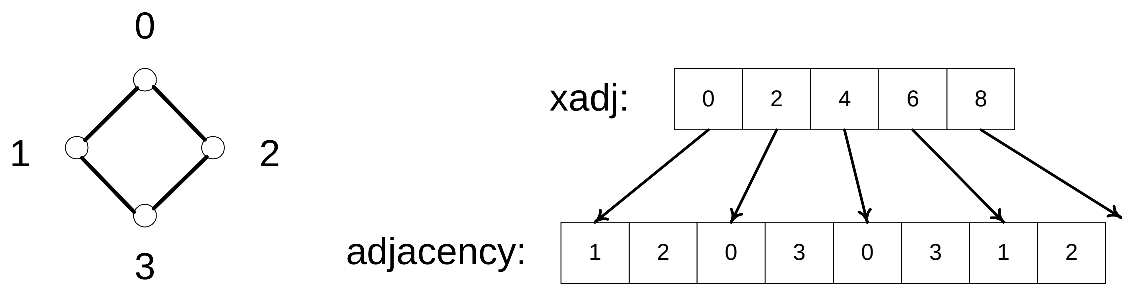
Model of Parallel Computation	**Edgecut should be minimal**		**Edgecut should be minimal**	
	Weights of edges are the same	**Weights of edges are different**	**Weights of the vertices are the same**	**Weights of the vertices are different**
Speeds are functions of the size of the problem and no limits exist on number of elements that each processor can hold.	No known results	No known results	No known results	No known results
Speeds are single constant numbers and an upper bound exists on number of elements that each processor can hold.	No known results	No known results	No known results	No known results
Speeds are single constant numbers and no limits exist on number of elements that each processor can hold.	No known results	No known results	No known results	No known results

Table 4.7: Special cases of partitioning of a Graph.

varying processing power and an underlying non-uniform communication network. Their partitioning algorithm generates and maps partitions onto a heterogeneous algorithm with the objective of minimizing the maximum execution time of the parallel application.

Table 4.7 summarizes specific partitioning problems for a graph and its current state to the best knowledge of the authors.

The basic approach to dealing with graph partitioning is to construct an initial partition of the vertices according to some problem-specific strategy such as given by criterion (a) or (b) and such that a vertex and as far as possible all its neighbors belong to a same partition. Then the



(a) (b)

Figure 4.20: (a) A sample graph, and (b) The adjacency structure of the graph shown in (a). Adjacency structure for vertex 0 starts at 0 and the number of adjacent vertices are 2 given by xadj[1] – xadj[0]. The adjacent vertices of vertex 0 are 1 and 2 given by adjacency[0] and adjacency[1]. Similarly adjacency structure for vertex 1 starts at 2 and the number of adjacent vertices are 2 given by xadj[2] – xadj[1]. The adjacent vertices are 0 and 3 given by adjacency[2] and adjacency[3].

vertices are swept one by one. A vertex is retained in the same partition if more of its neighbors given by its adjacency list are in the same partition. Otherwise the vertex is migrated to other partitions such that the edgecut is decreased. It is recommended that adjacency structure of a graph should have a specific structure, that is, the first adjacency list should correspond to vertex 0, the second adjacency list should correspond to the first neighbor of vertex 0 and so on. Also it is recommended that the numbering of vertices should follow a specific order. That is supposing the starting vertex *u* has a neighboring vertex *v* and vertex *v* has two neighbors, which are vertices w_1 and w_2. Then vertex *u* should be numbered 0 followed by 1 for its first and only neighbor *v*. The neighbors of *v*, w_1 and w_2, get the numbers 2 and 3 respectively. This is illustrated in Figure 4.20(b) showing the adjacency structure for a sample graph shown in Figure 4.20(a).

Based on this classification, we suggest an API that the application programmers can use to partition a **graph** into **p** disjoint partitions.

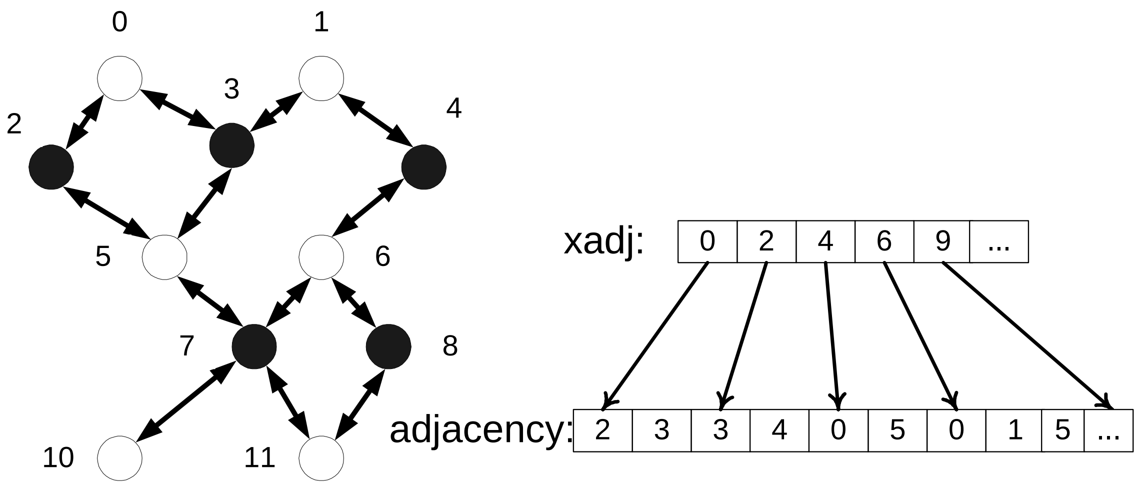
```
int Partition_graph (
     int p, int pn, const double *speeds,
     const int *psizes, const int *mlimits, int n, int m,
     const int *vwgt, const int *xadj,
     const int *adjacency, const int *adjwgt,
     int nopts, const int *options,
     int *vp, int *edgecut)
```

Parameter **`p`** is the number of partitions of the graph. Parameters **`speeds`** and **`psizes`** specify speeds of processors for **`pn`** different problem sizes. These parameters are 1D arrays of size **`p×pn`** logically representing 2D arrays of shape **`[p][pn]`**. The speed of the **`i`**-th processor for **`j`**-th problem size is given by the **`[i][j]`**-th element of **`speeds`** with the problem size itself given by the **`[i][j]`**-th element of **`psizes`**. Parameter **`mlimits`** gives the maximum number of elements that each processor can hold.

The parameters **`n`** and **`m`** are the number of vertices and edges in the graph. The parameters **`vwgt`** and **`adjwgt`** are the weights of vertices and edges of the graph. In the case in which the graph is unweighted (i.e., all vertices and/or edges have the same weight), then either or both of the arrays **`vwgt`** and **`adjwgt`** can be set to **`NULL`**. The parameters **`vwgt`** is of size **`n`**. The parameter **`adjwgt`** is of size **`2m`** because every edge is listed twice (i.e., as (v, u) and (u, v)).

The parameters **`xadj`** and **`adjacency`** specify the adjacency structure of the graph represented by the compressed storage format (CSR). The adjacency structure of the graph is stored as follows. The adjacency list of vertex **`i`** is stored in **`adjacency`** starting at index **`xadj[i]`** and ending at but not including **`xadj[i+1]`**. The adjacency lists for each vertex are stored consecutively in the array **`adjacency`**. Figure 4.20(b) shows the adjacency structure for a sample graph shown in Figure 4.20(a).

If the parameter **`vwgt`** is set to **`NULL`** and the processor speeds **`speeds`** are set to **`NULL`**, then the graph is partitioned into **`p`** disjoint partitions such that criterion (e) is satisfied. If the



(a) (b)

Figure 4.21: (a) A sample bipartite graph showing dependencies between black nodes and white nodes, and (b) The adjacency structure of the graph shown in (a).

parameter **`vwgt`** is set to **`NULL`** and the processor speeds **`speeds`** are not set to **`NULL`**, then the graph is partitioned into **`p`** disjoint partitions such that criterion (c) is satisfied. If the parameter **`vwgt`** is not set to **`NULL`** and the processor speeds **`speeds`** are not set to **`NULL`**, then the graph is partitioned into **`p`** disjoint partitions such that criterion (d) is satisfied.

The parameter **`options`** is an array of size **`nopts`** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes. For example, the partitioning schemes such as METIS [KK95], and Chaco [HL94] employ multilevel strategies consisting of various phases and heuristics are employed for every phase.

The parameter **`vp`** is an array of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which vertex **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

There are other types of graphs, whose partitionings are popularly used to solve parallel problems in scientific and engineering domains such as the bipartite graph [HK00] and hypergraph [CA96, PCA+96].

A **bipartite graph** $\boldsymbol{G} = (\boldsymbol{R}, \boldsymbol{C}, \boldsymbol{E})$ is a special type of graph in which the vertices are divided into two disjoint subsets, $\boldsymbol{R}$ and $\boldsymbol{C}$ and $\boldsymbol{E} \subset \boldsymbol{R} \times \boldsymbol{C}$. So, no edge connects vertices in the same subset; instead all the edges cross between $\boldsymbol{R}$ and $\boldsymbol{C}$.

The main criteria used for partitioning a bipartite graph are outlined below:

a) The number of vertices in each partition should be proportional to the speed of the processor owning that partition.

b) Each disjoint subset is partitioned such that the number of vertices in each partition should be proportional to the speed of the processor owning that partition.

c) The sum of weights of the vertices in each partition should be proportional to the speed of the processor owning that partition.

d) Each disjoint subset is partitioned such that the sum of weights of the vertices in each partition should be proportional to the speed of the processor owning that partition.

e) Set of disjoint partitions satisfying criterion (a) and the edgecut should be minimal. Edgecut is defined as the total number of edges in the graph whose incident vertices belong to different partition.

f) Set of disjoint partitions satisfying criterion (b) and the edgecut should be minimal. Edgecut is defined as the total number of edges in the graph whose incident vertices belong to different partition.

g) Set of disjoint partitions satisfying criterion (c) and the edgecut should be minimal. Edgecut is defined as the total weight of the edges in the graph whose incident vertices belong to different partition.

h) Set of disjoint partitions satisfying criterion (d) and the edgecut should be minimal. Edgecut is defined as the total weight of the edges in the graph whose incident vertices belong to different partition.

The partitioning operations on sets can be used to partition a bipartite graph such that either of criterion (a), (b), (c) and (d) is satisfied.

Application programmers can use the operation, whose interface is shown below, to partition a bipartite graph into **p** disjoint partitions.

```
int Partition_bipartite_graph (
   int p, int pn, const double *speeds,
   const int *psizes, const int *mlimits,
   int n, int m, const int *vtype, const int *vwgt,
   const int *xadj, const int *adjacency,
   const int *adjwgt, int type_of_partitioning,
   int nopts, const int *options,
   int *vp, int *edgecut)
```

The meaning of the parameters **p**, **pn**, **speeds**, **psizes**, **mlimits**, **n**, **m**, **vwgt**, **adjwgt**, **xadj**, **adjacency** is identical to meaning of the corresponding parameters of **Partition_graph**.

The parameter **vtype** specifies the type of vertex. The only values allowed are 0 and 1 representing the two disjoint subsets the bipartite graph is composed of. Figure 4.21(b) shows the adjacency structure for a sample bipartite graph shown in Figure 4.21(a).

The parameter **type_of_partitioning** specifies whether the partitioning of subsets is done separately or not. If the partitioning of subsets is to be done separately, then each subset is partitioned such that one of the criteria (a) or (b) is satisfied and edgecut is minimal. It can take only one of the values **`PARTITION_SUBSET`** and **`PARTITION_OTHER`**.

The parameter **`options`** is an array of size **`nopts`** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which vertex **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

A **hypergraph**, $\boldsymbol{H} = (\boldsymbol{V}, \boldsymbol{N})$, consists of a set of vertices, $\boldsymbol{V}$, and a set of hyperedges, $\boldsymbol{N}$. Each hyperedge comprises a subset of vertices. Let c_j denote the cost of hyperedge n_j. In a partition $\prod$ of hypergraph $\boldsymbol{H}$, a hyperedge that has atleast one vertex in that partition is said to *connect* that partition. Connectivity set $\boldsymbol{\Lambda}_j$ of a hyperedge n_j is defined as the set of partitions connected by n_j. Connectivity $\lambda_j = |\boldsymbol{\Lambda}_j|$ of a hyperedge n_j denotes the number of partitions connected by n_j. A hyperedge n_j is said to cut if it connects more than one partition, and uncut otherwise. The cut and uncut partitions are referred to as external nets and internal nets, respectively. The set of external nets of a partition $\prod$ is denoted as $\boldsymbol{N_E}$. Two relevant *cutsize* definitions are:

$$\text{(a) } \chi(\prod) = \sum_{n_\mathrm{j} \in \boldsymbol{N_E}} c_j \quad \text{and} \quad \text{(b) } \chi(\prod) = \sum_{n_\mathrm{j} \in \boldsymbol{N_E}} c_j(\lambda_j - 1)$$

The main criteria to partition the hypergraph $\boldsymbol{H}$ into $\boldsymbol{P}$ disjoint partitions include the main criteria used to partition a normal graph except that instead of satisfying the criterion that the edgecut should be minimal, the criterion that cutsize should be minimized is satisfied during the partitioning.

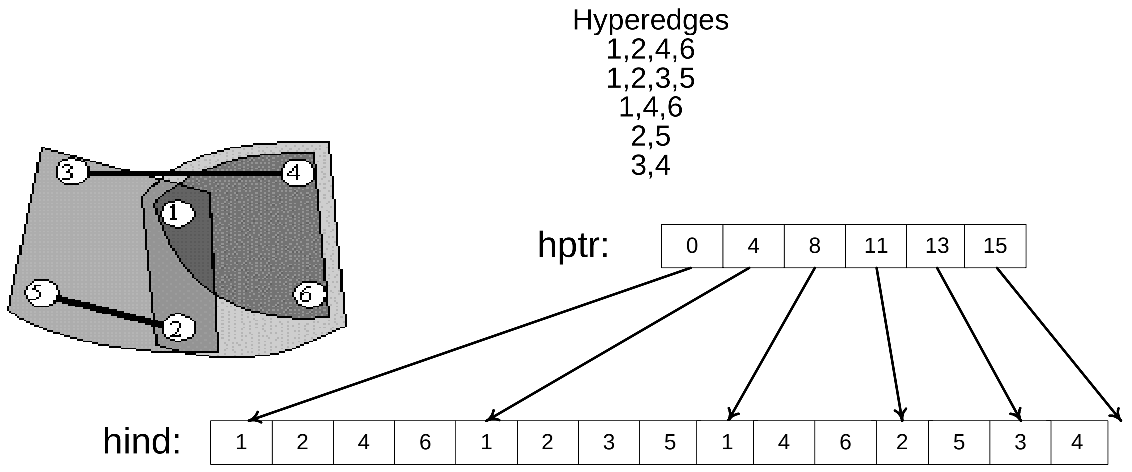


Figure 4.22: The **`hptr`** and **`hind`** arrays that are used to describe the hyperedges of the hypergraph.

Application programmers can use the operation, whose interface is shown below, to partition a **hypergraph** into **`p`** disjoint partitions.

```
int Partition_hypergraph (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits,
    int nv, int nedges, const int *vwgt, const int *hptr,
    const int *hind, const int *hwgt,
    int nopts, const int *options,
    int *vp, int *edgecut)
```

The meaning of the parameters **`p`**, **`pn`**, **`speeds`**, **`psizes`**, and **`mlimits`** is identical to meaning of the corresponding parameters of **`Partition_graph`**.

The parameters **`nv`** and **`nedges`** are the number of vertices and number of hyperedges in the hypergraph.

The parameters **`vwgt`** is an array of size **`nv`** that stores the weights of the vertices and **`hwgt`** is an array of size **`nedges`** that stores the weights of hyperedges of the graph. If the vertices in the hypergraph are unweighted, then **`vwgt`** can be **`NULL`**. If the hyperedges in the hypergraph are unweighted, then **`hwgt`** can be **`NULL`**.

The parameter **`hptr`** is an array of size **`nedges`**`+1` and is an index into **`hind`** that stores the actual hyperedges. Each hyperedge stores the sequence of the vertices that it spans, in consecutive locations in **`hind`**. Specifically, **`i`**-th hyperedge is stored starting at location **`hind[hptr[i]]`** up to but not including **`hind[hptr[i+1]]`**. Figure 4.22 illustrates the format for a simple hypergraph.

The parameter **`options`** is an array of size **`nopts`** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes. For example, the partitioning schemes such as hMETIS ([KAK+97], [KK98b]) employ multilevel strategies consisting of various phases and heuristics are employed for every phase.

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which vertex **`i`** belongs to. The parameter **`edgecut`** contains the number of hyperedges that are cut by the partitioning.

4.6 Partitioning Problems for Trees

A **tree** is a graph such that there is a unique simple path between each pair of vertices. There are five main criteria in partitioning a tree into a set of disjoint subtrees:

a) The number of elements in each subtree should be proportional to the speed of the processor owning that subtree.

b) The sum of weights of elements in each subtree should be proportional to the speed of the processor owning that subtree.

c) Set of disjoint partitions satisfying criterion (a) and the edgecut should be minimal. Edgecut is defined as the total weight of the edges in the tree whose incident vertices belong to different subtrees.

d) Set of disjoint partitions satisfying criterion (b) and the edgecut should be minimal.

e) The edgecut should be minimal.

The implicit restriction is that the tree should be partitioned into disjoint subtrees such that one of the above criteria is satisfied. The partitioning operations on graphs can be used to partition a tree into disjoint partitions when there is no restriction that all the disjoint partitions have to be subtrees. Additional restrictions that may be imposed are that the number of vertices in each partition must be less than the maximum number of elements a processor can hold.

The partitioning operations on sets can be used to partition a tree into **p** disjoint partitions such that either of criteria (a) or (b) is satisfied.

Application programmers can use the operation, whose interface is shown below, to partition a tree into **p** disjoint partitions.

```
int Partition_tree (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits,
    int n, int nedges, const int *nwgt, const int *xadj,
    const int *adjacency, const int *adjwgt,
    int *vp, int *edgecut)
```

The meaning of the parameters **p**, **pn**, **speeds**, **psizes**, and **mlimits** is identical to meaning of the corresponding parameters of **Partition_graph**.

The parameters **n** and **nedges** are the number of vertices and edges in the tree. The parameters **nwgt** is an array of size **n** that stores the weights of the vertices and **adjwgt** is an array of size **nedges** that stores the weights of edges of the tree. If the vertices in the tree are unweighted, then **`nwgt`** can be **`NULL`**. If the edges in the tree are unweighted, then **`adjwgt`** can be **`NULL`**.

The parameters **`xadj`** and **`adjacency`** specify the adjacency structure of the tree.

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which node **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

HMPI provides an additional operation, which allows the application programmer to formulate the heuristic solutions for their optimization problems used to partition a tree.

4.7 Algorithms of Partitioning Sets

In this section, we present the algorithms of partitioning sets. In the figures we present for illustration, we use the notion of problem size. Kumar *et al*. [KGG+94] define the problem size as the number of basic computations in the best sequential algorithm to solve the problem on a single processor. Because it is defined in terms of sequential time complexity, the problem size is a function of the size of the input. For example, the problem size is $O(n^3)$ for $\mathbf{n}\times\mathbf{n}$ matrix multiplication and for irregular applications such as EM3D [YWC+95, CDG+93] and N-body simulation [BN97], the problem size is O(n), where **n** is the number of nodes in a bipartite graph representing the dependencies between the nodes and number of bodies respectively.

However we do not use this computational complexity definition for problem size because it does not influence the speed of the processor. We define the size of the problem to be the amount of data stored and processed by the sequential algorithm. For example for matrix-matrix multiplication of two dense $\mathbf{n}\times\mathbf{n}$ matrices, the size of the problem is equal to $3\times\mathbf{n}^2$.

To demonstrate the efficiency of our data partitioning algorithms using the functional model, we perform experiments using naïve parallel algorithms for linear algebra kernel, namely, matrix multiplication and LU factorization using striped partitioning of matrices on a local network of heterogeneous computers. Our main aim is not to show how matrices can be efficiently multiplied or efficiently factorized but to explain in simple terms how the data partitioning algorithms using the functional model can be used to optimally schedule arbitrary tasks on networks of heterogeneous computers before moving on to solve the most advanced problem. We also view these algorithms as good representatives of a large class of data parallel computational problems and a good testing platform before experimenting with more challenging computational problems.

4.7.1 Algorithms for Partitioning Sets without Processor Memory Bounds

In this section, we solve the following problem of partitioning a set, which can be formulated as:

Given: (1) A set of **n** elements, and (2) A well-ordered set of **p** heterogeneous processors whose speeds are functions of the size of the problem, $\mathbf{s_i=f_i(x)}$, and (3) There is no upper bound on the largest problem size that can be solved on each processor;

Partition the set into **p** disjoint partitions such that:

- $x_0+x_1+...+x_{\mathbf{p}-1}=\mathbf{n}$, where $x_0,x_1,...,x_{\mathbf{p}-1}$ are the number of elements in partitions 0,1,…,**p**-1 respectively;
- $\frac{x_0}{s_0}=\frac{x_1}{s_1}=...=\frac{x_i}{s_i}=...=\frac{x_{p-1}}{s_{p-1}}$ where $x_0,x_1,...,x_{\mathbf{p}-1}$ are the number of elements assigned to the processors 0,1,…,**p**-1 respectively and $s_0,s_1,...,s_{\mathbf{p}-1}$ are the speeds of the processors.

We provide an optimal solution to this problem of complexity $O(\mathbf{p}^2\times\log_2\mathbf{n})$.

One of the criteria to partitioning a set of **n** elements over **p** heterogeneous processors is that the number of elements in each partition should be proportional to the speed of the processor owning that partition. When the speed of the processor is represented by a single number, the algorithm used to perform the partitioning is quite straightforward, of complexity $O(\mathbf{p}^2)$ [BBP+01]. The algorithm uses a naive implementation. The complexity can be reduced down to $O(\mathbf{p}\times\log_2\mathbf{p})$ using ad hoc data structures [BBP+01].

This problem of partitioning a set becomes non-trivial when the speeds of the processors are given as a function of the size of the problem. Consider a small network of two processors, whose speeds as functions of problem size during the execution of the matrix-matrix multiplication are shown in Figure 4.23. If we use the single number model, we have to choose a point and use the absolute speeds of the processors at that point to partition the elements of the

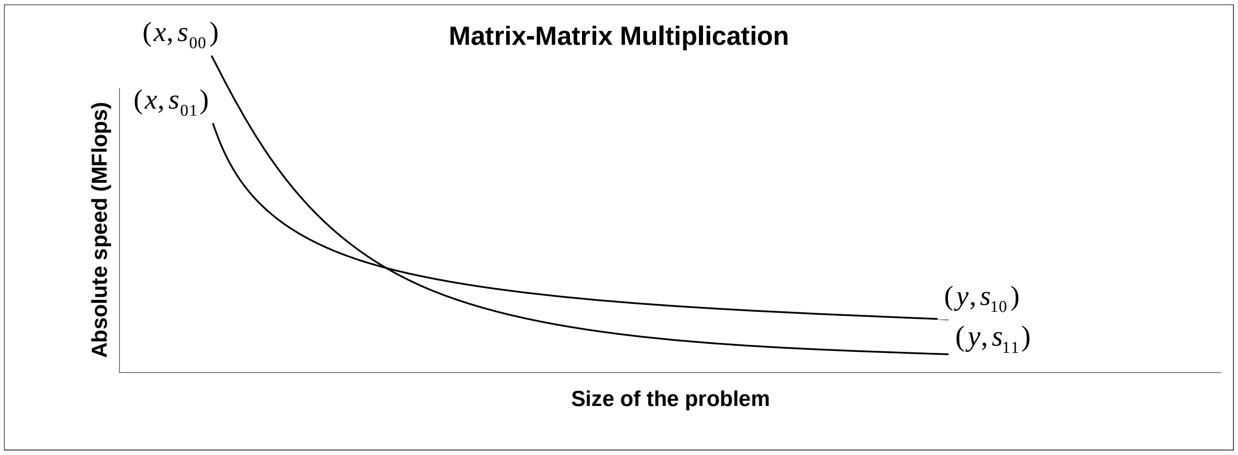


Figure 4.23: A small network of two processors whose speeds are shown against the size of the problem. The Matrix-Matrix Multiplication used here uses a poor solver that does not use memory hierarchy efficiently.

set such that the number of elements is proportional to the speed of the processor. If we choose the speeds (s_{00}, s_{01}) at points (x, s_{00}) and (x, s_{01}) to partition the elements of the set, the distribution obtained will be unacceptable for the size of the problem at points (y, s_{10}) and (y, s_{11}) where processors demonstrate different relative speeds compared to the relative speeds at points (x, s_{00}) and (x, s_{01}). If we choose the speeds (s_{10}, s_{11}) at points (y, s_{10}) and (y, s_{11}) to partition the elements of the set, the distribution obtained will be unacceptable for the size of the problem at points (x, s_{00}) and (x, s_{01}) where processors demonstrate different relative speeds compared to the relative speeds at points (y, s_{10}) and (y, s_{11}). In some such cases, the partitioning of the set obtained could be the worst possible distribution where the number of elements per processor obtained could be inversely proportional to the speed of the processor. In such cases, it is better to use an even distribution of equal number of elements per processor than the distribution based on using such wrong points.

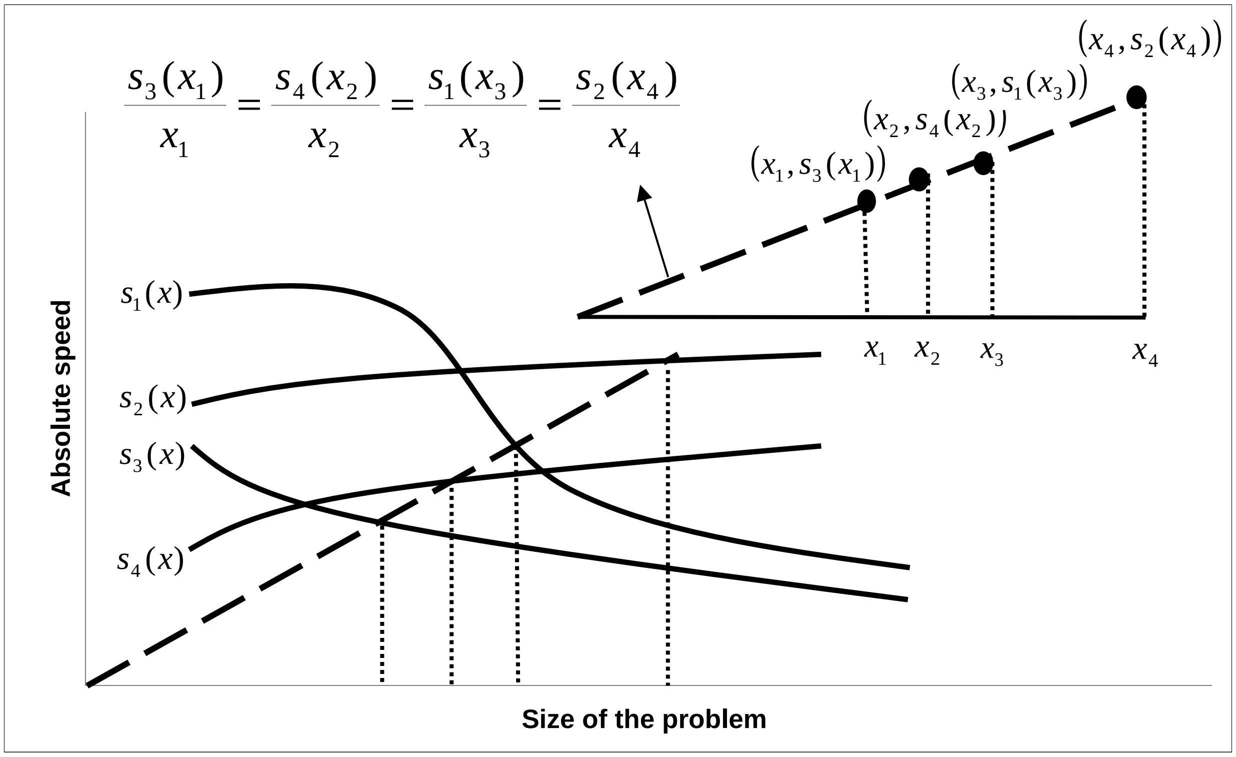


Figure 4.24: Optimal solution showing the geometric proportionality of the number of elements to the speed of the processor. s_1(x), s_2(x), s_3(x), and s_4(x) are speeds of processors 1, 2, 3, and 4 respectively, which are functions of the size of the problem.

The algorithms we propose are based on the following observation: If a distribution of the elements of the set amongst the processors is obtained such that the number of elements is proportional to the speed of the processor, then the points, whose coordinates are number of elements and speed, lie on a straight line passing through the origin of the coordinate system and intersecting the graphs of the processors with speed versus the size of the problem in terms of the number of elements. This is shown by the geometric proportionality in Figure 4.24.

Our general approach to finding the optimal straight line can be summarized as follows:

1. We assume that the speed of each processor is represented by a continuous function of the size of the problem. The shape of the graph should be such that there is only one intersection

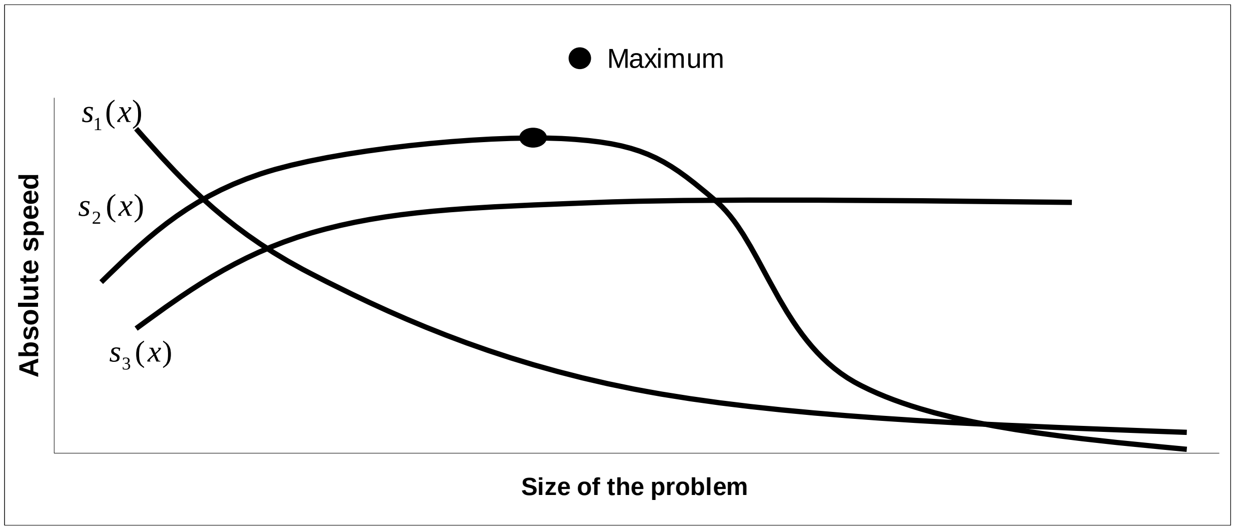


Figure 4.25: Typical shapes of the graphs representing the speed functions of the processors observed experimentally. The graph represented by $s_1(x)$ is strictly a decreasing function of the size of the problem. The graph represented by $s_2(x)$ is initially an increasing function of the size of the problem followed by a decreasing function of the size of the problem. The graph represented by $s_3(x)$ is strictly an increasing function of the size of the problem.

point of the graph with any straight line passing through the origin. These assumptions on the shapes of the graph are representative of the most general shape of graphs observed for applications experimentally. The experiments conducted by Lastovetsky and Twamley [LT04] justify these assumptions. Applications that utilize memory hierarchy efficiently and applications that reference memory randomly deriving no benefits from caching produce speed functions that are an increasing function of problem size before a maximum followed by a decreasing function of problem size whereas applications that use inefficient memory reference patterns produce speed functions that are strictly decreasing functions of problem size. Some of the sample shapes of the graphs are shown in Figure 4.25.

2. At each step, we have two lines both passing through the origin. The sum of the number of elements at the intersection points of the first line with the graphs is less than the size of the problem, and the sum of the number of elements at the intersection points of second line with the graphs is greater than the size of the problem.
3. The region between these two lines is divided by a line passing through the origin into two smaller regions, the upper region and the lower region. If the sum of the number of elements at the intersection points of this line with the graphs is less than the size of the problem, the optimal line lies in the lower region. If this sum is greater than the size of the problem, the optimal line lies in the upper region.
4. In general, the exact optimal line intersects the graphs in points with non-integer sizes of the problem. This line is only used to obtain an approximate integer-valued solution. Therefore, the finding of any other straight line, which is close enough to the exact optimal one to lead to the same approximate integer-valued solution, will be an equally satisfactory output of the searching procedure. A simple stopping criterion for this iterative procedure can be the absence of points of the graphs with integer sizes of the problem within the current region. Once the stopping region is reached, the two lines limiting this region are input to the fine tuning procedure, which determines the optimal line.

Note that it is the continuity and the shape of the graphs representing the speed of the processors that make each step of this procedure possible. The continuity guarantees that any straight line passing through the origin will have at least one intersection point with each of the graphs, and the shape of the graph guarantees no more than one such an intersection point.

We now prove the uniqueness of the solution using mathematical induction starting by illustrating with an example for **p**=3. We safely assume that for each processor, for all $x \geq y$,

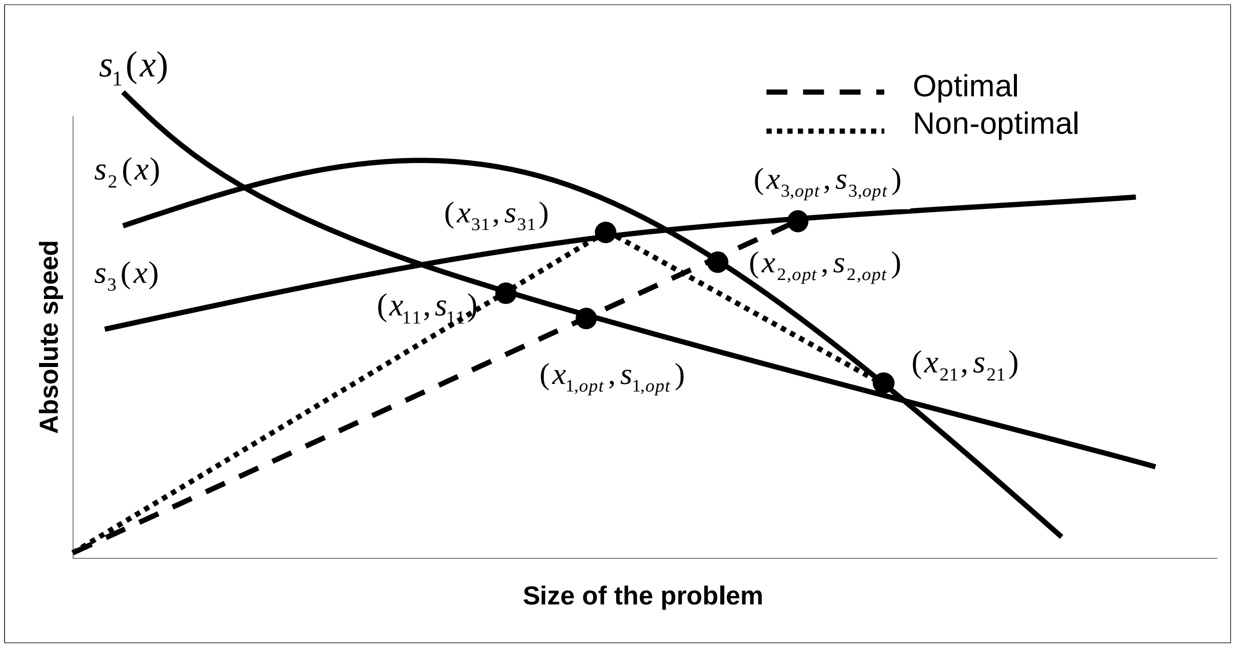


Figure 4.26: Uniqueness of the solution. The dashed line represents the optimal solution whereas the dotted line represents a non-optimal solution.

where x and y are problem sizes, the execution times t_x and t_y to execute problems of sizes x and y respectively are related by $t_x \geq t_y$. We also assume that the volume of computations involved in the execution of a problem size is equal to the problem size. Consider a small network of three processors, whose speeds as functions of problem size are shown in Figure 4.26. The graph represented by $s_1(x)$ is strictly a decreasing function of the size of the problem. The graph represented by $s_2(x)$ is initially an increasing function of the size of the problem followed by a decreasing function of the size of the problem. The graph represented by $s_3(x)$ is strictly an increasing function of the size of the problem. We show two solutions for a problem size **n**. The non-optimal solution is given by (x_{11},x_{21},x_{31}) such that $x_{11}+x_{21}+x_{31}=$**n** and the optimal solution is given by $(x_{1,opt},x_{2,opt},x_{3,opt})$ such that $x_{1,opt}+x_{2,opt}+x_{3,opt}=$**n**. The time of execution for the optimal solution t_{opt} is $(x_{1,opt}/s_{1,opt})$ or $(x_{2,opt}/s_{2,opt})$ or $(x_{3,opt}/s_{3,opt})$ because $(x_{1,opt}/s_{1,opt})=(x_{2,opt}/s_{2,opt})=(x_{3,opt}/s_{3,opt})$. The time of execution of the non-optimal solution is

$\max\limits_{i=1}^{3}(\frac{x_{i1}}{s_{i1}})$. Since $x_{21}>x_{2,opt}$, we can conclude that time of execution $t_{21}$of the problem size x_{21} equal to x_{21}/s_{21} is always greater than the time of execution given by the optimal solution t_{opt} i.e., $t_{21}>t_{opt}$. Thus it can be inferred that the time of execution of the application obtained using the non-optimal solution is always greater than or equal to the time of execution of the application using the optimal solution. We can easily prove the same for different shapes of the speed functions.

Assuming this to be true for **p**=**k** processors, we have to prove the optimality for **p**=**k+1** processors. For a given problem size **n**, let us assume the distribution given by our algorithm to be $(x_{1,opt},x_{2,opt},\ldots,x_{\mathbf{k+1},opt})$ such that $(x_{1,opt}/s_{1,opt})=(x_{2,opt}/s_{2,opt})=\ldots=(x_{\mathbf{k+1},opt}/s_{\mathbf{k+1},opt})=t_{opt}$ and $x_{1,opt}+x_{2,opt}+\ldots+x_{\mathbf{k+1},opt}=\mathbf{n}$ where t_{opt} is the time of execution of the algorithm by the optimal solution. Now consider a distribution $(x'_1,x'_2,..,x'_{\mathbf{k+1}})$ such that $x'_1+x'_2+\ldots+x'_{\mathbf{k+1}}=\mathbf{n}$ and $x'_i \neq x_{i,opt}$ for all i=1,2,…,**k+1**. If $(x'_1/s'_1)=(x'_2/s'_2)=\ldots=(x'_{\mathbf{k+1}}/s'_{\mathbf{k+1}})=t'_e$, then it can be inferred that if $t'_e=t_{opt}$, then $x'_i>x_{i,opt}$ or $x'_i<x_{i,opt}$ for all i=1,2,…,**k+1** in which case the equality $x'_1+x'_2+\ldots+x'_{k+1}=\mathbf{n}$ is broken. If the proportionality $(x'_1/s'_1)=(x'_2/s'_2)=\ldots=(x'_{\mathbf{k+1}}/s'_{\mathbf{k+1}})$ is ignored but the equality $x'_1+x'_2+\ldots+x'_{\mathbf{k+1}}=\mathbf{n}$ is satisfied, then it can be easily seen that for atleast one processor i (i=1,2,…,**k+1**), $x'_i>x_{i,opt}$, thus giving an execution time t'_i, which is greater than the execution time given by our algorithm t_{opt}. It is easy to extend this proof for cases where the volume of computations performed by the processor is proportional to the problem size assigned to it.

Without loss of generality, in the figures we show the application of the algorithm only in the regions where absolute speed is a decreasing function of the size of the problem.

Let us estimate the cost of one step of this procedure. At each step we need to find the points of intersection of **p** graphs $\mathbf{y=s_1(x)}$, $\mathbf{y=s_2(x)}$, ..., $\mathbf{y=s_p(x)}$, representing the absolute speeds of the

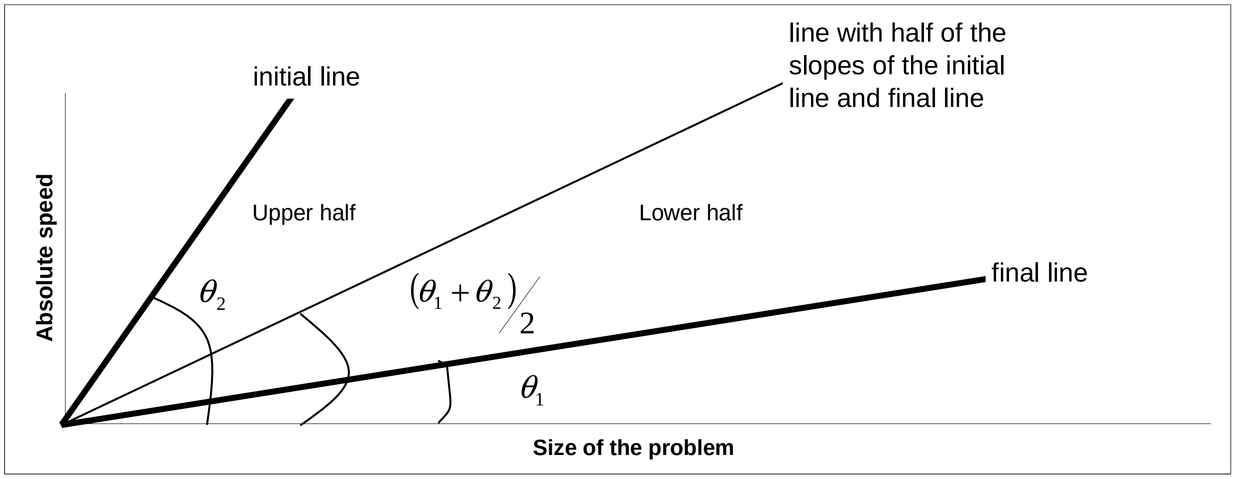


Figure 4.27: Determination of the slope of the line equal to half of the slopes of the initial and final lines.

processors, and the straight line **y=c×x** passing through the origin. In other words, at each step we need to solve **p** equations of the form **c×x =s$_1$(x)**, **c×x =s$_2$(x)**, ..., **c×x =s$_p$(x)**. As we need the same constant number of operations to solve each equation, the complexity of this part of one step will be O(**p**). According to our stopping criterion, a test for convergence can be reduced to testing **p** inequalities of the form **l$_i$ - u$_i$** <1, where **l$_i$** and **u$_i$** are the size coordinates of the intersection points of the **i**-th graph with the lower and upper lines limiting the region respectively (**i**=1,2,…,**p**). This testing is also of the complexity O(**p**). Therefore, the total complexity of one step including the convergence test will still be O(**p**).

The simplest particular algorithm based on this approach bisects the region between the lines by a line passing through the origin at a slope equal to half of the sum of the slopes of the two lines as shown in Figure 4.27. These slopes are angles and not the tangent of the angles. However in practical implementations of the algorithm, slopes that are tangents can be used instead of angles for efficiency from computational point of view.

The use of bisection is shown in Figure 4.28. The first two lines drawn during step 1 are **`line1`** and **`line2`**. Then **`line3`** is drawn whose slope is half of the slopes of the lines **`line1`**

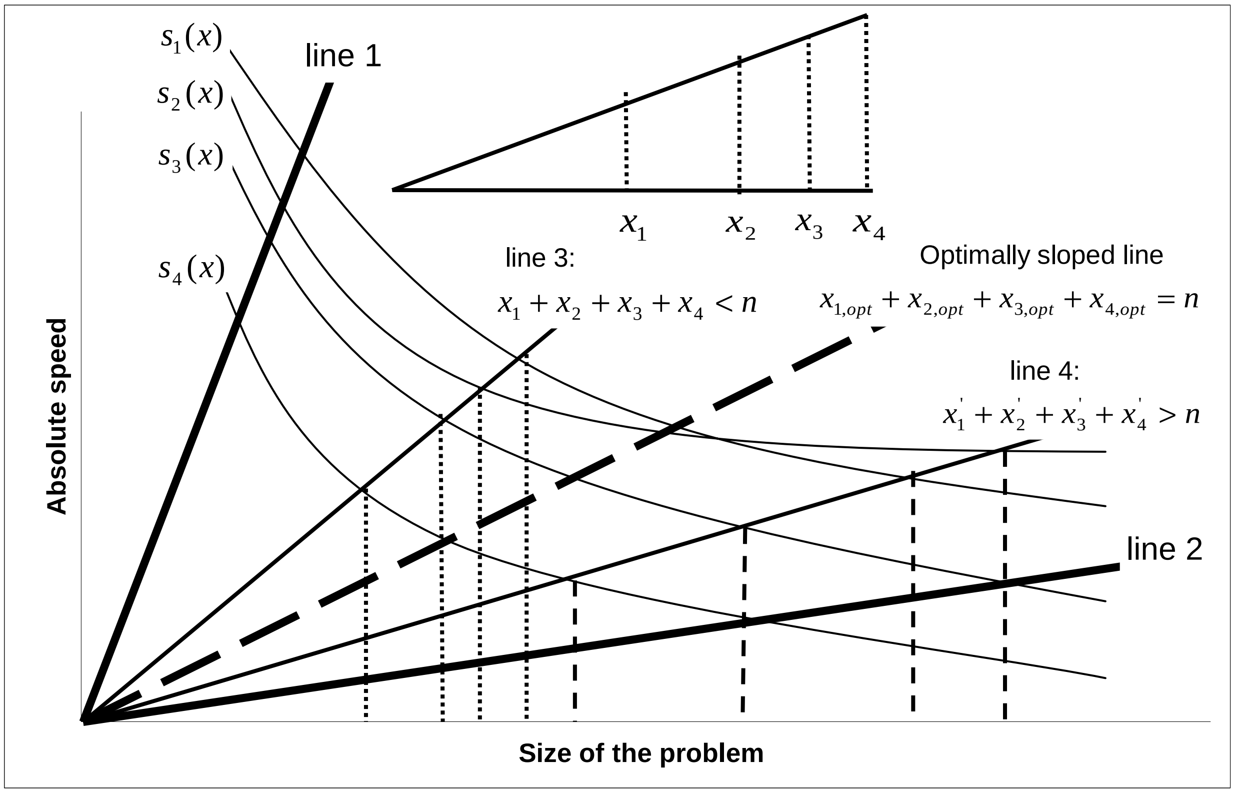


Figure 4.28: Use of bisection of the range to narrow down to the optimal solution satisfying the criterion that the number of elements should be proportional to the speed of the processor. **n** is the size of the problem.

and **`line2`**. Since the sum of the number of elements at the intersection points of this line with the graphs is less than the size of the problem, bisect the lower half of the region by drawing **`line4`** whose slope is half of the slopes of the lines **`line3`** and **`line2`**. Since the sum of the number of elements at the intersection points of this line with the graphs is greater than the size of the problem, bisect the upper half of the region by drawing a line whose slope is half of the slopes of the lines **`line3`** and **`line4`**. This line turns out to be the optimally sloped line.

In most real-life situations, this algorithm will demonstrate a very good efficiency. Obviously, the slope of the optimal line is a decreasing function of the size of the problem, $\theta_{opt} = \theta_{opt}(\mathbf{n})$. If $\theta_{opt}(\mathbf{n}) = O(\mathbf{n}^{-\mathbf{k}})$, where **k**=const, then the maximal number of steps to arrive at the

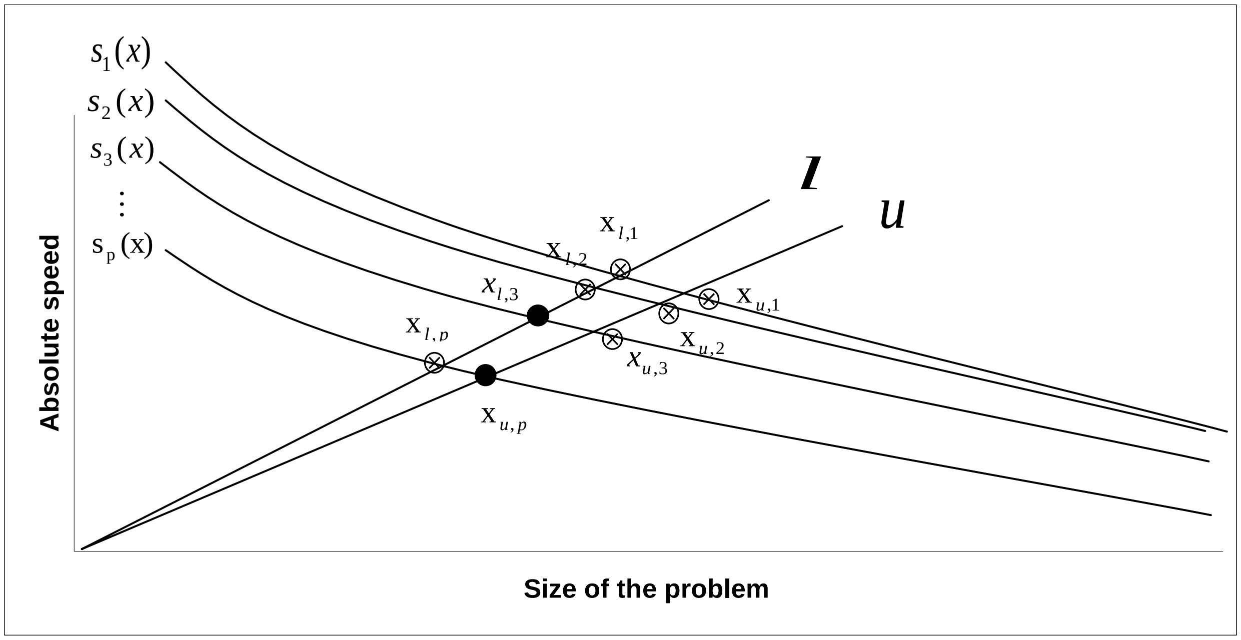


Figure 4.29: Fine tuning procedure chooses the final **p** points of intersection from the **p** integer points closest to the non-integer points on line **l** and **p** integer points closest to the non-integer points on line **u**. There are no integers between the lines **l** and **u**. The integers closest to the non-integer points on lines **l** and **u** are indicated by crossed dots whereas integer points lying on lines **l** and **u** are indicated by dark dots.

stopping criterion will be O(**k**×$\log_2$**n**). Correspondingly, the complexity of the algorithm up till this point will be O(**p**×$\log_2$**n**).

Once we have reached the stopping criterion indicated by the absence of points of the graphs with integer sizes of the problem within the region bounded by lines **l** and **u**, we perform additional fine tuning to find the **p** integer points on the curves representing the speed functions of the processors thus giving us a solution closest to the optimal non-integer solution. This is illustrated in Figure 4.29. As can be seen from the figure, there are **2×p** points, **p** integer points ($x_{\mathbf{l},1}$,$x_{\mathbf{l},2}$,…,$x_{\mathbf{l},\mathbf{p}}$), some of which could be closest to the non-integer points on line **l** whereas the rest of them lying on line **l** and similarly **p** integer points ($x_{\mathbf{u},1}$,$x_{\mathbf{u},2}$,…,$x_{\mathbf{u},\mathbf{p}}$) pertaining to line **u**.

We have to choose an optimal set of **p** points from these **2×p** points. The fine tuning procedure consists of the following steps:

1. We find out the times of execution $\mathbf{x_i}/\mathbf{s_i}$ of the problem sizes at these **2×p** points where $\mathbf{x_i}$ is the problem size assigned to the processor **i** and $\mathbf{s_i}$ is its speed exhibited at this problem size. This step is of complexity O(**p)**.
2. We then sort these **2×p** execution times using Quicksort algorithm and choose the **p** best execution times. The complexity of the Quicksort algorithm is $O(\mathbf{2}\times\mathbf{p}\times\log_2(\mathbf{2}\times\mathbf{p})) = O(\mathbf{p}\times\log_2\mathbf{p})$.

The total complexity of the fine tuning process is $O(\mathbf{p})+O(\mathbf{p}\times\log_2\mathbf{p}) = O(\mathbf{p}\times\log_2\mathbf{p})$. So the total complexity of our partitioning algorithm is given by $O(\mathbf{p}\times\log_2\mathbf{n})+O(\mathbf{p}\times\log_2\mathbf{p}) = O(\mathbf{p}\times\log_2(\mathbf{n}\times\mathbf{p}))$. If $\mathbf{n} \gg \mathbf{p}$, the total complexity of our partitioning algorithm is given by $O(\mathbf{p}\times\log_2\mathbf{n})$.

At the same time, in some situations this algorithm may be quite expensive. For example, if $\theta_{opt}(\mathbf{n})=O(e^{-\mathbf{n}})$, then the number of steps to arrive at the optimal line will be $O(\mathbf{n})$. Correspondingly, the complexity of the algorithm will be $O(\mathbf{p}\times\mathbf{n})$. After fine tuning, the complexity of the algorithm will be $O(\mathbf{p}\times\mathbf{n})+O(\mathbf{p}\times\log_2\mathbf{p}) = O(\mathbf{p}\times\mathbf{n})$.

We modify this algorithm to achieve reasonable performance in all cases, independent on how the slope of the optimal line depends on the size of the problem. To introduce the modified algorithm, let us re-formulate the problem of finding the optimal straight line as follows:

1. The space of solutions consists of all straight lines drawn through the origin and intersecting the graphs of the processors so that the size coordinate of at least one intersection point is integer.
2. We search for a straight line from this space closest to the optimal solution.

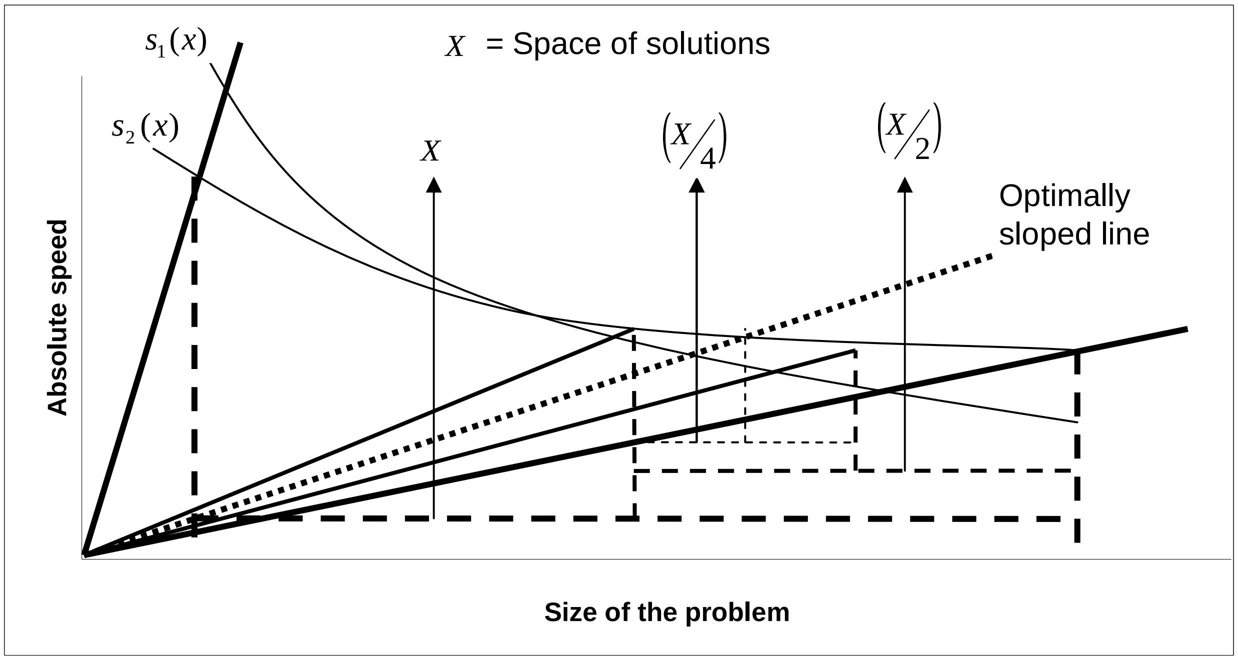


Figure 4.30: Bisection of the space of solutions in the modified algorithm.

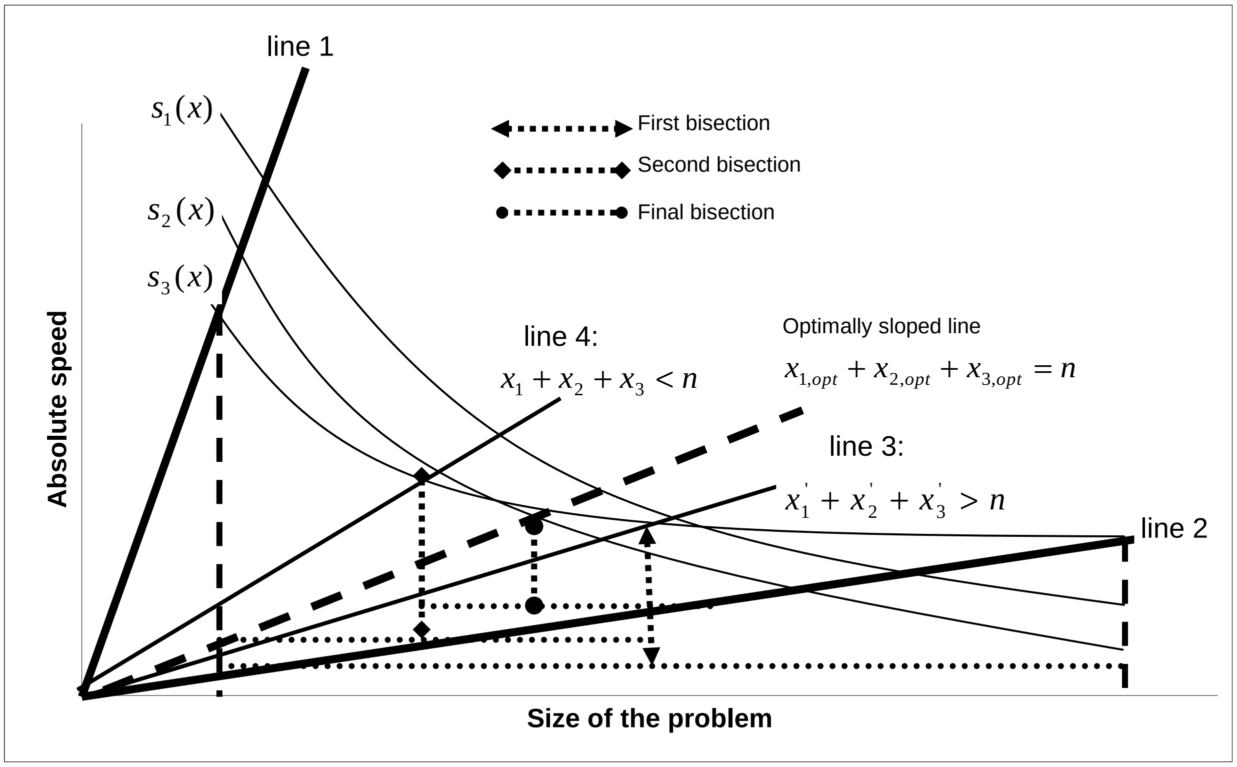


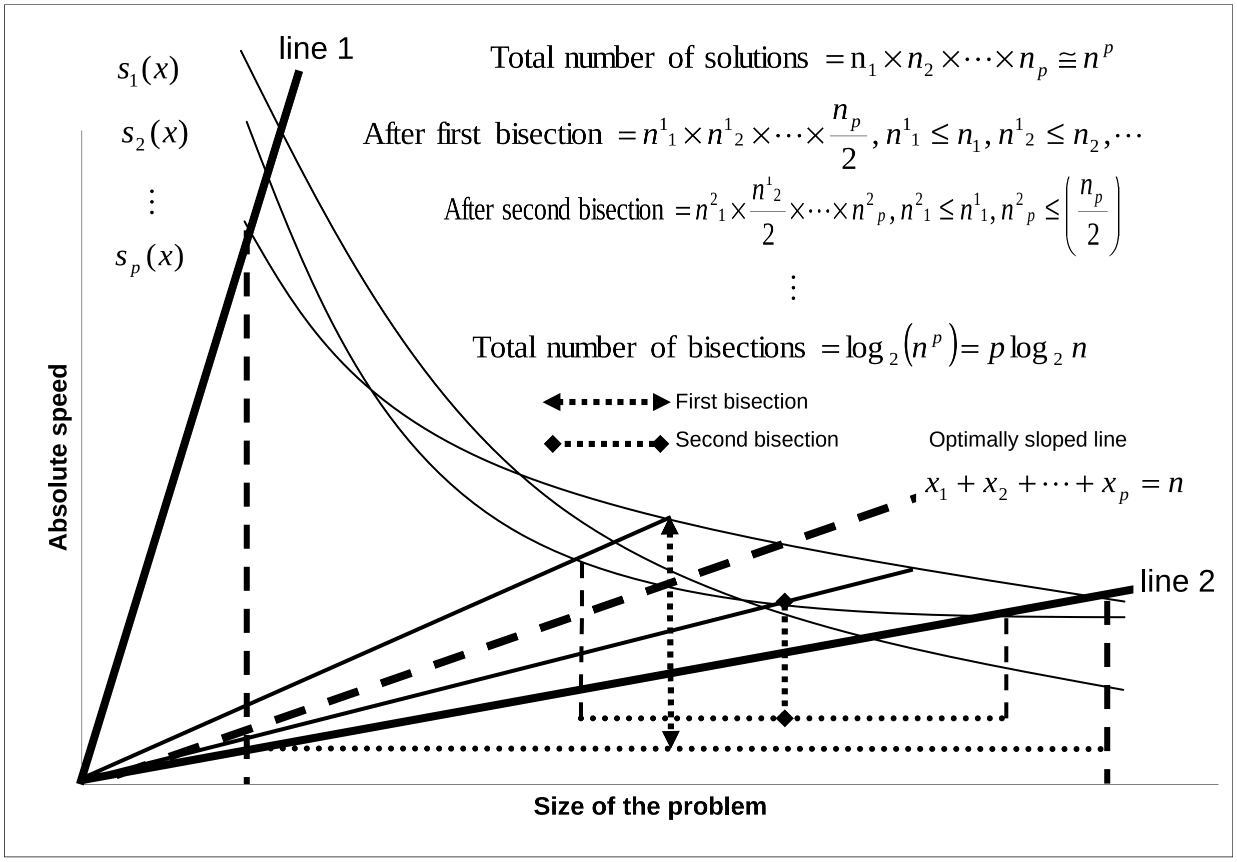
Figure 4.31: Modification of the algorithm shown in Figure 4.28 where the bisection results in efficient solution. **n** is the size of the problem.

At each step of the basic bisection algorithm, it is the region between two lines that is reduced, not the space of solutions. Our modified algorithm tries to reduce the space of solutions rather than the region where the solution lies as illustrated in Figure 4.30 and Figure 4.31. At each step of the algorithm, we find a processor, whose graph **s(x)** is intersected by the maximal number of lines from the current region of the space of solutions limited by the lower and upper lines. Then we detect a line, which divides the region into two smaller regions such that each region contains the same number of lines from the space of solutions intersecting this graph. To do it, we just need to draw a line passing through the origin and the point ((**v**-**w**)/2, **s**((**v**-**w**)/2)), where **v** and **w** are the size coordinates of the intersection points of this graph with the lower and upper lines limiting the current region of the space of solutions.

This algorithm guarantees that after **p** such bisections the number of solutions in the region is reduced at least by 50%. This means we need no more than $\mathbf{p}\times\log_2\mathbf{n}$ steps to arrive at the sought line. Correspondingly, the complexity of this algorithm will be $O(\mathbf{p}^2\times\log_2\mathbf{n})$. After fine tuning, the complexity of the algorithm will be $O(\mathbf{p}^2\times\log_2\mathbf{n})+O(\mathbf{p}\times\log_2\mathbf{p}) = O(\mathbf{p}^2\times\log_2\mathbf{n})$. A schematic proof of the algorithm is shown in Figure 4.32.

One can see that the modified bisection algorithm is not sensitive to the shape of the graphs of the processors, always demonstrating the same efficiency. The basic bisection algorithm is sensitive to their shape. It demonstrates higher efficiency than the modified one in better cases but much lower efficiency in worse cases.

An ideal bisection algorithm would be of the complexity $O(\mathbf{p}\times\log_2\mathbf{n})$ reducing at each step the space of solutions by 50% and being insensitive to the shape of the graphs of the processors. The design of such an algorithm is still a challenge.



$s_1(x)$
$s_2(x)$
$\vdots$
$s_p(x)$
line 1
Total number of solutions $= n_1 \times n_2 \times \cdots \times n_p \cong n^p$
After first bisection $= n^1{}_1 \times n^1{}_2 \times \cdots \times \frac{n_p}{2}, n^1{}_1 \le n_1, n^1{}_2 \le n_2, \cdots$
After second bisection $= n^2{}_1 \times \frac{n^1{}_2}{2} \times \cdots \times n^2{}_p, n^2{}_1 \le n^1{}_1, n^2{}_p \le \left(\frac{n_p}{2}\right)$
Total number of bisections $= \log_2(n^p) = p\log_2 n$
First bisection
Second bisection
Optimally sloped line
$x_1 + x_2 + \cdots + x_p = n$
line 2
Absolute speed
Size of the problem

Figure 4.32: Schematic proof of the complexity of the modified algorithm. The total number of bisections is **p×log$_2$n**. At each step of bisection, **p** intersection points are obtained giving a total complexity of **O(p^2×log$_2$n)**.

In cases where the magnitude of the size of the problem is of order millions, it might be worth relaxing the stopping criterion and not using the fine-tuning procedure. However it should be noted that the complexity of the algorithm will remain the same as fine-tuning procedure does not add to the overall complexity although the cost in practice is minimized by relaxing the stopping criterion. If all the sub-optimal solutions are close to each other as to be indistinguishable as in this case, we can provide an approximate solution that is sufficiently accurate and at the same time economical in terms of practical cost. We intend to investigate this

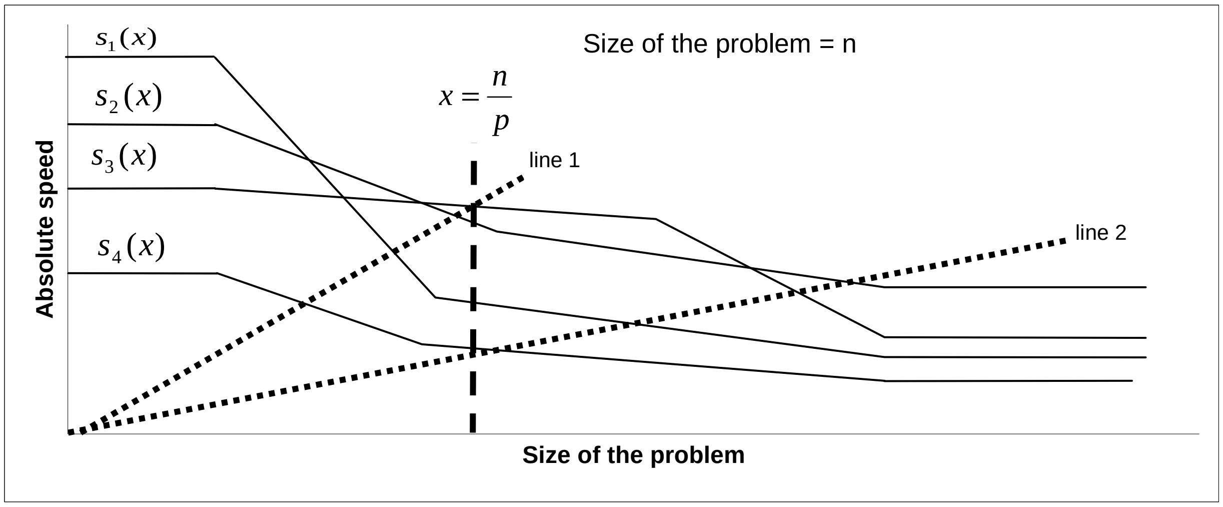


Figure 4.33: For most real-life situations, the optimal solution lies in the region with polynomial slopes. The optimal solution lies between **line1** and **line2** and they enclose a region with all polynomial slopes.

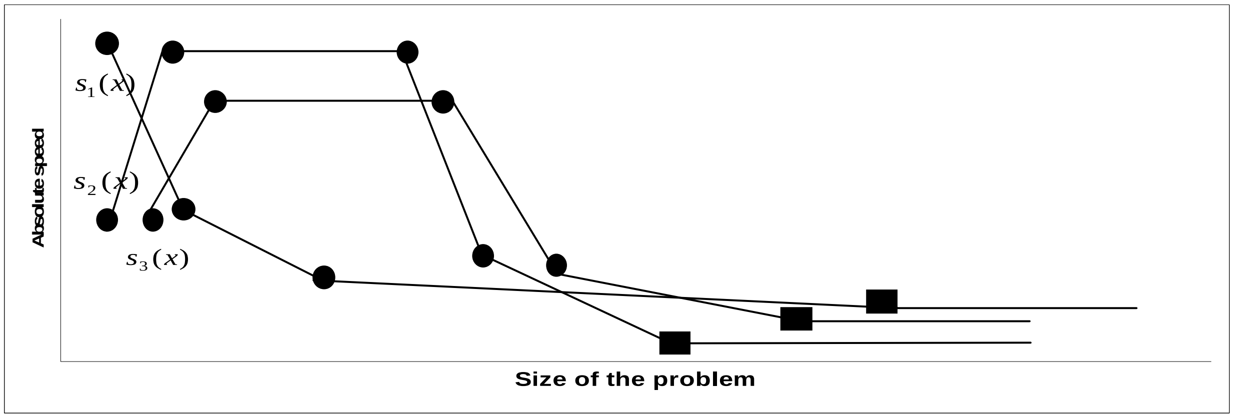


Figure 4.34: Using piecewise linear approximation to build speed functions for 3 processors. The circular points are experimentally obtained whereas the square points are calculated using heuristics but not experimentally obtained. The speed function for processor $s_1(x)$ is built from 3 experimentally obtained points (application run on this processor uses memory hierarchy inefficiently) whereas the speed functions $s_2(x)$ and $s_3(x)$ (application run on these processors use memory hierarchy efficiently) are built from 4 experimentally obtained points.

further in future research to provide an approximate solution that maintains a balance between accuracy and economy.

For a large range of problem sizes, it is very likely that the optimal solutions lie in the region with polynomial slopes as shown in Figure 4.33. In these cases, the simplest algorithm gives the optimal solution with best efficiency. However for very large problem sizes where the shapes of the speed functions tend to be horizontal, the modified algorithm gives the optimal solution with best efficiency. However both the simplest and the modified algorithm can be combined to solve data partitioning problems in real-life applications efficiently.

One approach consists of the following steps:

1. Speed functions are built for the processors involved in the execution of the parallel application using a set of few experimentally obtained points. One of the ways to build a speed function for a processor is to use piecewise linear function approximation as shown in Figure 4.34. Such approximation of the speed function is compliant with the requirements of the functional model. Also such an approximation of the speed function should give the speed of the processor for a problem size within acceptable limits of deviation from the speed given by an ideal speed function or the speed functions built with sets with more number of points. A practical procedure to build this piecewise linear function approximation of speed function is explained in detail in Section 4.1.1.
2. Having built the speed functions for the processors, we use the simplest algorithm to bisect the region between the lower and upper lines as shown in Figure 4.35. If the solution lies in the upper half and the line bisecting the region between the lower and the upper lines intersects the graphs of the processors at polynomial slopes, we use the simplest algorithm to obtain the optimal solution. This is because we know that the speed functions have polynomial slopes in the upper region and in such a case the simplest algorithm gives an optimal solution with ideal complexity. In other cases such as when the solution lies in the

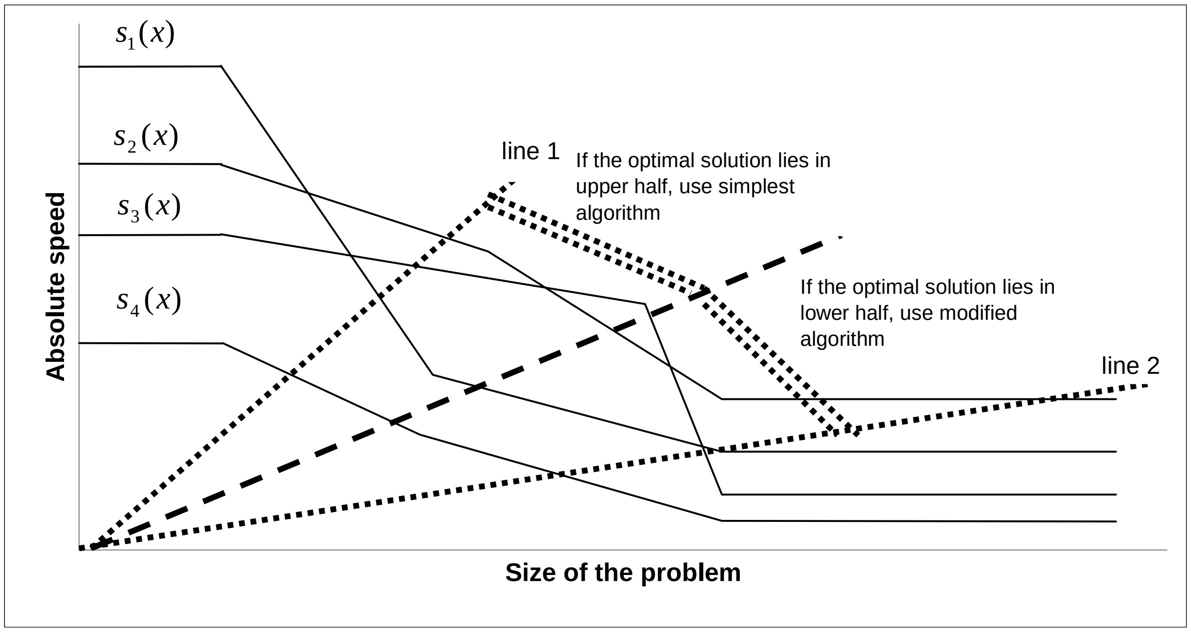


Figure 4.35: Using a combination of simplest and modified algorithm to efficiently solve problems for real-life applications.

upper half and if the line bisecting the region between the lower and the upper lines intersects one or more graphs at horizontal slope or when the solution lies in the lower half, we use modified algorithm to obtain the optimal solution. This is because in such a case, we know that the modified algorithm is proven to demonstrate better efficiency than the simplest algorithm.

In case of large problem sizes where the application slows down to a considerable extent due to severe paging, it is advisable to use out-of-core algorithms [Tol99].These algorithms are designed to achieve high performance when their data structures are stored on disks. When an algorithm is to be executed out-of-core, the ordering of independent operations must be chosen so as to minimize I/O. In addition, the layout of data structures on disks must be chosen so that I/O is performed in large blocks, and so that all or most of the data that is read in one block-I/O operation is used before it is evicted from main memory. Our data partitioning algorithms using

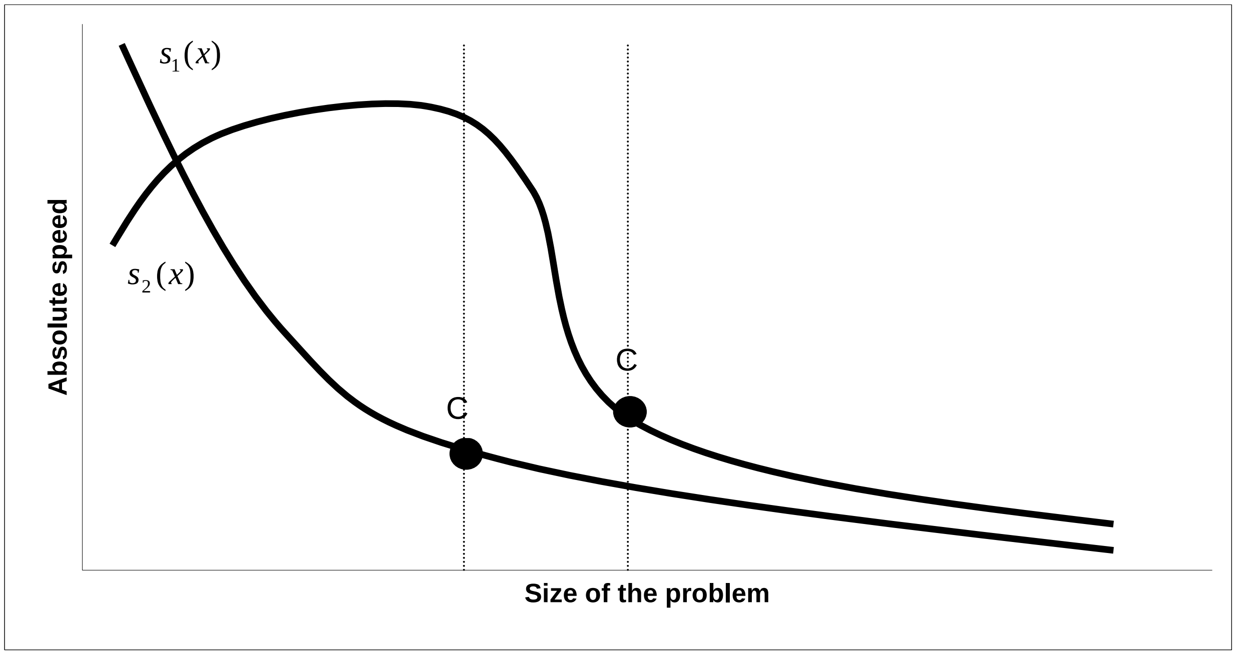


Figure 4.36: Illustration of the regions where our data partitioning algorithms are better compared to out-of-core algorithms and vice versa. C is the crossover point at which out-of-core algorithms start performing better than our data partitioning algorithms.

the functional model of networks of heterogeneous computers gives better results when applied to regions left of the crossover point C as shown in Figure 4.36. Out-of-core algorithms perform better than our data partitioning algorithms in regions to the right of crossover point C. We aim to research further to find this crossover point C with minimal experimental time.

4.7.1.1 Experimental Results

The experimental results are divided into two sections. The first section is devoted to building the functional model. We present the parallel applications and the network of heterogeneous computers on which the applications are tested. For each application, we explain how to estimate

Machine Name	Architecture	cpu MHz	Total Main Memory (kBytes)	Available Main Memory (kBytes)	Cache (kBytes)	Paging (MM)	Paging (LU)
X1	Linux 2.4.20-20.9 i686 Intel Pentium III	997	513304	363264	256	4500	6000
X2	Linux 2.4.18-3 i686 Intel Pentium III	997	254576	65692	256	4000	5000
X3	Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM)	2783	7933500	2221436	512	6400	11000
X4	Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM)	2783	7933500	3073628	512	6400	11000
X5	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	415904	512	6000	8500
X6	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	364120	512	6000	8500
X7	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	215752	512	6000	8000
X8	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	134400	512	5500	6500
X9	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	134400	512	5500	6500
X10	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	409600	2048	4500	5000
X11	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	418816	2048	4500	5000
X12	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	395264	2048	4500	5000

Table 4.8: Specifications of the twelve heterogeneous processors to demonstrate the efficiency of the functional performance model. Paging is the size of the matrix beyond which point paging started happening.

the processor speed. The procedure to build the speed functions of the processors is explained in section 4.1.1. For each application, we determine the problem sizes beyond which point paging starts happening and we also give the cost of building the speed function of each processor. We discuss the cost involved in finding the optimal solution using the partitioning algorithm and find it negligible compared to the execution time of the applications which varies from minutes to hours. In the second section, we present the experimental results obtained by running these applications on the network of heterogeneous computers.

4.7.1.1.1 Applications

A small heterogeneous local network of 12 different Solaris and Linux workstations shown in Table 4.8 is used in the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers. The amount of memory, which is the difference between the total main memory and available main memory shown in the tables, is used by the operating system processes and few other user application processes that perform routine computations and communications such as email clients, browsers, text editors, audio applications etc. These processes use a constant percentage of CPU.

There are two applications used to demonstrate the efficiency of our data partitioning algorithms using the functional model.

Matrix-matrix multiplication

The first application shown in Figure 4.37(a) multiplies matrix **A** and matrix **B**, i.e., implementing matrix operation $\mathbf{C}=\mathbf{A}\times\mathbf{B}^{\mathbf{T}}$, where A, B, and C are dense square $\mathbf{n}\times\mathbf{n}$ matrices. The application uses a parallel algorithm of matrix-matrix multiplication of two dense matrices using horizontal striped partitioning [Las03, p.199], which is based on a heterogeneous 1D clone of the

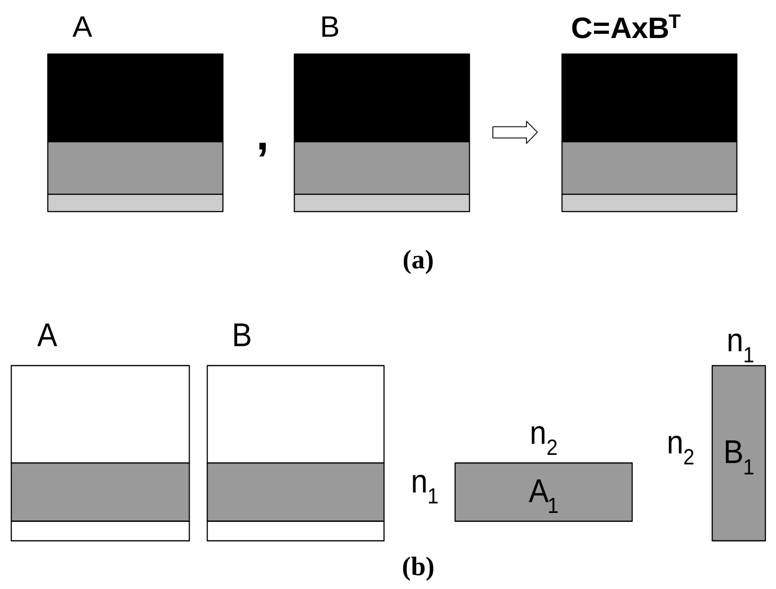


Figure 4.37: (a) Matrix operation $C=A\times B^T$ with matrices A, B, and C. Matrices A, B, and C are horizontally sliced. The number of elements in each slice is proportional to the speed of the processor. (b) Serial matrix multiplication $A_1\times B_1$ ($B_1=B^T$) of two dense non-square matrices of sizes $n_1\times n_2$ and $n_2\times n_1$ respectively to estimate the absolute speed of processor 1. The parameter n_2 is fixed during the application of the set partitioning algorithm and is equal to **n**.

parallel algorithm used in ScaLAPACK [CDD+96] for matrix multiplication. The matrices A, B, and C are partitioned into horizontal slices such that the total number of elements in the slice is proportional to the speed of the processor.

For the application implementing matrix operation $\mathbf{C}=\mathbf{A}\times\mathbf{B}^\mathbf{T}$, the absolute speed of a processor must be obtained based on multiplication of two dense non-square matrices of sizes $\mathbf{n_1}\times\mathbf{n_2}$ and $\mathbf{n_2}\times\mathbf{n_1}$ respectively as illustrated in Figure 4.37(b). Even though there are two parameters $\mathbf{n_1}$ and $\mathbf{n_2}$ representing the size of the problem, the parameter $\mathbf{n_2}$ is fixed and is equal to **n** during the application of the set partitioning algorithm. To apply the set partitioning algorithm to determine the optimal data distribution for such an application, we need to extend it for problem size represented by two parameters, $\mathbf{n_1}$ and **n**. The speed function of a processor is geometrically a surface when represented by a function of two parameters $\mathbf{s=f(n_1,n_2)}$. However since the parameter $\mathbf{n_2}$ is fixed and is equal to $\mathbf{n}$, the surface is reduced to a line $\mathbf{s=f(n_1,n_2)=}$ $\mathbf{s=f(n_1,n)}$. Thus the set partitioning problem for this application reduces to the algorithm that we have presented in section 4.7.1. However additional computations are involved in obtaining experimentally the geometric surfaces representing the speed functions of the processors and then reducing them to lines.

Our algorithm of partitioning of a set can be extended easily to obtain optimal solutions for problem spaces with two or more parameters representing the problem size. Each such problem space is reduced to a problem formulated using a geometric approach and tackled by extensions of our geometric set-partitioning algorithm. Consider for example the case of two parameters representing the problem size where neither of them is fixed. In this case, the speed functions of the processors are represented by surfaces. The optimal solution provided by a geometric algorithm would divide these surfaces to produce a set of rectangular partitions equal in number to the number of processors such that the number of elements in each partition (the area of the partition) is proportional to the speed of the processor. We do not present the extensions of our algorithm here for such multi-dimensional representations of the size of the problem. We think it would complicate the presentation.

To calculate the absolute speed of the processor, we use a serial version of the parallel algorithm of matrix-matrix multiplication. The serial version performs matrix-matrix multiplication of two dense square matrices. Though the absolute speed must be obtained by multiplication of two dense non-square matrices, we observed that our serial version gives almost the same speeds for multiplication of two dense square matrices if the number of elements in a dense non-square matrix is the same as the number of elements in a dense square

Size of matrix	Absolute speed (MFlops)	Size of matrix	Absolute speed (MFlops)	Size of matrix	Absolute speed (MFlops)	Size of matrix	Absolute speed (MFlops)
256×256	67	1024×1024	67	2304×2304	67	4096×4096	59
128×512	68	512×2048	66	1152×4608	67	2048×8192	60
64×1024	67	256×4096	67	576×9216	69	1024×16384	59
32×2048	67	128×8192	67	288×18432	70	512×32768	60

Table 4.9: Results of serial matrix-matrix multiplication to demonstrate the effect of the number of elements in a matrix on the absolute speed of the processor.

matrix. This is illustrated in Table 4.9 for one Linux computer X8 whose specification is shown in Table 4.8. The behavior exhibited is the same for other computers. Thus speed functions of the processors built using dense square matrices will be the same as those built using dense non-square matrices.

LU Factorization

The second application is based on the parallel algorithm of LU factorization of a dense square **n**×**n** matrix **A**, one step of which is shown in Figure 4.38(a). On a homogeneous **p**-processor linear array, a CYCLIC(**b**) distribution of columns is used to distribute the matrix *A* where **b** is the block size [CDO+96, BBP+01]. A cyclic distribution would assign block numbers 0,1,2,…,**n**-1 to processor 0,1,2,…,**p**-1,0,1,2…,**p**-1,0,…, respectively, for a **p**-processor linear array (**n**»**p**), until all **n** blocks are assigned. At each step of the algorithm, the processor that owns the pivot block factors it and broadcasts it to all the processors, which update their remaining blocks. At the next step, the next block of **b** columns becomes the pivot panel, and the computation progresses. Figure 4.38(a) shows how the column panel, L_{11} and L_{21}, and the row panel, U_{11} and U_{12}, are computed and how the trailing submatrix A_{22} is updated. Because the largest fraction of the work takes place in the update of A_{22}, therefore, to obtain maximum parallelism all processors should participate in the updating. Since A_{22} reduces in size as the

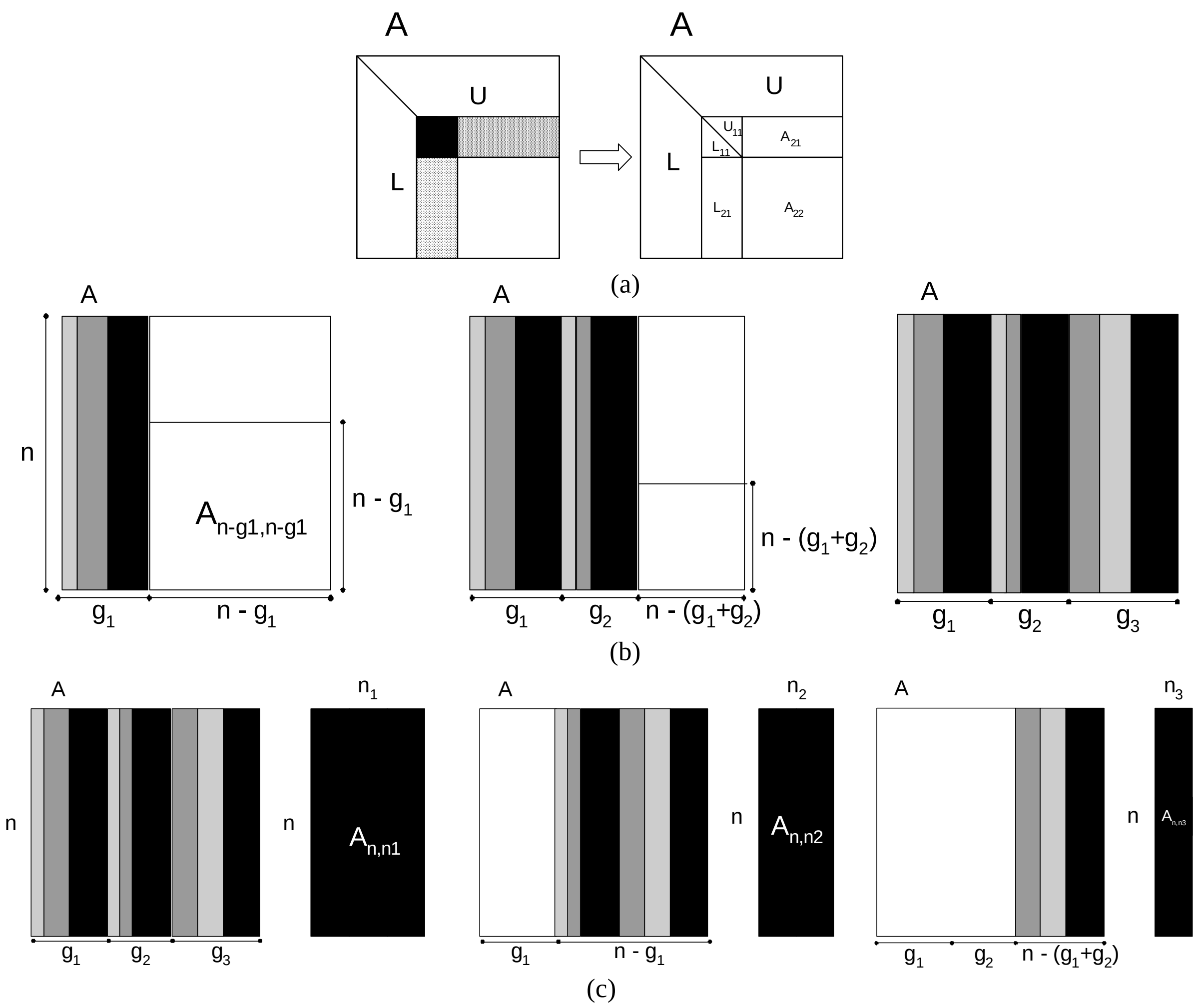


Figure 4.38: (a) One step of the LU factorization algorithm of a dense square matrix A of size **n**×**n**. (b) The matrix A is partitioned using Variable Group Block distribution. This figure illustrates the distribution for **n**=576,**b**=32,**p**=3. The distribution inside groups G_1, G_2, and G_3 are {2,1,1,0,0,0}, {2,1,0,0,0}, and {2,2,1,1,0,0,0}. (b) Serial LU factorization of a dense non-square matrix is used to estimate the absolute speed of a processor. Since the Variable Group Block distribution uses the functional model where absolute speed of the processor is represented by a function of a size of the problem, the distribution uses absolute speeds at each step of the LU decomposition that are based on the size of the problem solved at that step. As seen in this figure, at each of the steps for processor 0, the functional dependence of the absolute speed on the problem size gives the speeds based on solving the problem size at that step, which is equal to the number of elements in matrices A_{n,n_1}, A_{n,n_2}, and A_{n,n_3} respectively. That is at each of the steps for processor 0, the absolute speeds are based on serial LU decomposition of matrices A_{n,n_1}, A_{n,n_2}, and A_{n,n_3}.

computation progresses, a cyclic distribution is used to ensure that at any stage A_{22} is evenly distributed over all processors, thus obtaining a balanced load.

Two load balancing algorithms, namely, Group Block algorithm [BTP00, BMP04] and Dynamic Programming algorithm [BBP+01] have been proposed to obtain optimal static distribution over **p** heterogeneous processors arranged in a linear array. The Group Block distribution partitions the matrix into groups, all of which have the same number of blocks. The number of blocks per group (size of the group) and the distribution of the blocks in the group amongst the processors are fixed and are determined based on speeds of the processors, which are represented by a single constant number. Same is the case with Dynamic Programming distribution except that the distribution of the blocks in the group amongst the processors is determined based on dynamic programming algorithm.

We propose a Variable Group Block distribution, which is a modification of the Group Block algorithm. It uses the functional model where absolute speed of the processor is represented by a function of a size of the problem. Since the Variable Group Block distribution uses the functional model where absolute speed of the processor is represented by a function of a size of the problem, the distribution uses absolute speeds at each step of the LU decomposition that are based on the size of the problem solved at that step. That is at each step, the number of blocks per group and the distribution of the blocks in the group amongst the processors are determined based on absolute speeds of the processors given by the functional model, which are based on solving the problem size at that step. Thus it also takes into account the effects of paging.

Figures 4.38(b) and 4.38(c) illustrate the Variable Group Block algorithm of a dense square **n**×**n** matrix *A* over **p** heterogeneous processors. Given a dense **n**×**n** square matrix *A* and a block size of **b**, the Variable Group Block distribution is a static data distribution that vertically partitions the matrix into **m** groups of blocks whose column sizes are $\mathbf{g_1, g_2, \ldots, g_m}$ as shown in Figure 4.38(b). The groups are non-square matrices of sizes $\mathbf{n\times(g_1\times b)}, \mathbf{n\times(g_2\times b)}, \ldots, \mathbf{n\times(g_m\times b)}$ respectively. The steps involved in the distribution are:

1). To calculate the size $\mathbf{g_1}$ of the first group G_1 of blocks, we adopt the following procedure:

- Using the data partitioning algorithm, we obtain an optimal distribution of matrix A such that the number of elements assigned to each processor is proportional to the speed of the processor. The optimal distribution derived is given by (x_i, s_i) $(0\le i\le \mathbf{p}-1)$, where x_i is the size of the subproblem such that $\sum_{i=0}^{p-1} x_i = n^2$ and s_i is the absolute speed of the processor used to compute the subproblem x_i for processor i. Calculate the load index $l_i = s_i \big/ \sum_{k=0}^{p-1} s_k$ $(0\le i\le \mathbf{p}-1)$.
- The size of the group $\mathbf{g_1}$ is equal to $\lfloor 1/\min(l_i) \rfloor$ $(0\le i\le \mathbf{p}-1)$. If $\mathbf{g_1}/\mathbf{p}<2$, then $g_1 = \lfloor 2/\min(l_i) \rfloor$. This condition is imposed to ensure there is sufficient number of blocks in the group.
- This group G_1 is now partitioned such that the number of blocks $g_{1,i}$ is proportional to the speeds of the processors s_i where $\sum_{i=0}^{p-1} g_{1,i} = g_1$ $(0\le i\le \mathbf{p}-1)$.

2). To calculate the size $\mathbf{g_2}$ of the second group, we repeat step 1 for the number of elements equal to $(\mathbf{n-g_1})^2$ in matrix *A*. This is represented by the sub-matrix $\mathbf{A_{n-g_1,n-g_1}}$ shown in Figure 4.38(b). We recursively apply this procedure until we have fully vertically partitioned the matrix *A*.

3). For algorithms such as LU Factorization, only blocks below the pivot are updated. The global load balancing is guaranteed by the distribution in groups; however, for the group that holds the pivot it is not possible to balance the workload due to the lack of data. Therefore it is possible to reduce the processing time if the last blocks in each group are assigned to fastest processors, that is when there is not enough data to balance the workload then it should be the fastest processors doing the work. That is in each group, processors are reordered to start from the slowest processors to the fastest processors for load balance purposes.

In LU Factorization, the size of the matrix shrinks as the computation goes on. This means that the size of the problem to be solved shrinks with each step. Consider the first step. After the factorization of the first block of **b** columns, there remain **n**-**b** columns to be updated. At the second step, the number of columns to update is only **n**-2×**b**. Thus the speeds of the processors to be used at each step should be based on the size of the problem solved at each step, which means that for the first step, the absolute speed of the processors calculated should be based on the update of **n**-**b** columns and for the second step, the absolute speed of the processors calculated should be based on the update of **n**-2×**b** columns. Since the Variable Group Block distribution uses the functional model where absolute speed of the processor is represented by a function of a size of the problem, the distribution uses absolute speeds at each step that are calculated based on the size of the problem solved at that step.

For the application implementing LU factorization, the absolute speed of a processor must be obtained based on LU factorization of a dense non-square matrix of size $\mathbf{m_1}\times\mathbf{m_2}$ as shown in Figure 4.38(c). Even though there are two parameters $\mathbf{m_1}$ and $\mathbf{m_2}$ representing the size of the problem, the parameter $\mathbf{m_1}$ is fixed and is equal to **n** during the application of the set partitioning algorithm. To apply the set partitioning algorithm to determine the optimal data distribution for such an application, we need to extend it for problem size represented by two parameters, **n** and $\mathbf{m_2}$. The speed function of a processor is geometrically a surface when represented by a function of two parameters $\mathbf{s=f(m_1,m_2)}$. However since the parameter $\mathbf{m_1}$ is fixed and is equal to **n**, the

Size of matrix	**Absolute speed (MFlops)**	**Size of matrix**	**Absolute speed (MFlops)**	**Size of matrix**	**Absolute speed (MFlops)**	**Size of matrix**	**Absolute speed(MFlops)**
1024×1024	115	2304×2304	129	4096×4096	131	6400×6400	132
512×2048	115	1152×4608	130	2048×8192	132	3200×12800	131
256×4096	116	576×9216	129	1024×16384	132	1600×25600	132
128×8192	117	288×18432	129	512×32768	131	800×51200	131

Table 4.10: Results of serial LU factorization to demonstrate the effect of the number of elements in a matrix on the absolute speed of the processor.

surface is reduced to a line $\mathbf{s=f(m_1,m_2)= s=f(n,m_2)}$. Thus the set partitioning problem for this application reduces to the algorithm that we have presented in section 4.7.1. However additional computations are involved in obtaining experimentally the geometric surfaces representing the speed functions of the processors and then reducing them to lines.

The set partitioning algorithm can also be extended here easily as explained for matrix multiplication. To calculate the absolute speed of the processor, we use a serial version of the parallel algorithm of LU factorization. The serial version performs LU factorization of a dense square matrix. Though the absolute speed must be obtained by using LU factorization of a dense non-square matrix, we observed that our serial version gives almost the same speeds for LU factorization of a dense square matrix if the number of elements in a dense non-square matrix is the same as the number of elements in a dense square matrix. This is illustrated in Table 4.10 for computer X8 whose specification is shown in Table 4.8. The behavior exhibited is the same for other computers.

The absolute speed of the processor in number of floating point operations per second is calculated using the formula

$$\text{Absolute speed} = \frac{\text{volume of computations}}{\text{time of execution}} = \frac{\text{MF} \times \text{n} \times \text{n} \times \text{n}}{\text{time of execution}}$$

where **n** is the size of the matrix. **MF** is 2 for Matrix Multiplication and 2/3 for LU factorization.

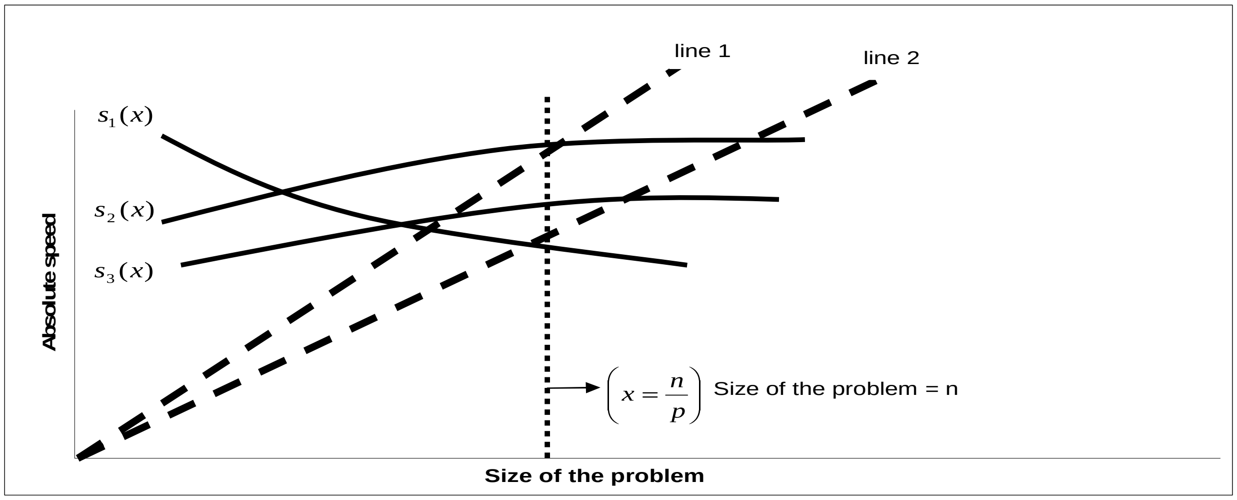


Figure 4.39: Detection of the initial two lines between which the solution lies.

The two lines **line1** and **line2**, between which the solution lies, are also inputs to the partitioning algorithms. We detect these lines as shown in Figure 4.39. Suppose the problem size is **n** and the number of processors involved in the execution of the problem size is **p**. Obtain the speeds of the processors with each processor executing a problem size of (**n**/**p**). The first line **line1** is drawn passing through the origin and a point, the coordinates of which are (**n**/**p**) and the highest speed. The second line **line2** is drawn passing through the origin and a point, the coordinates of which are (**n**/**p**) and the lowest speed.

For matrix-matrix multiplication, the computer X5 exhibited the fastest speed of 250 MFlops for multiplying two dense 4500×4500 matrices whereas the computer X10 exhibited the lowest speed of 31 MFlops at that problem size. The ratio $\frac{250}{31} \approx 8.0$ suggests that the processor set is reasonably heterogeneous. It should be noted that paging has not started happening at this problem size for both the computers. Similarly for LU factorization, the computer X6 exhibited the fastest speed of 130 MFlops for factorizing a dense 8500×8500 matrix whereas the computer X1 exhibited the lowest speed of 19 MFlops for factorizing a dense 4500×4500 matrix. The ratio

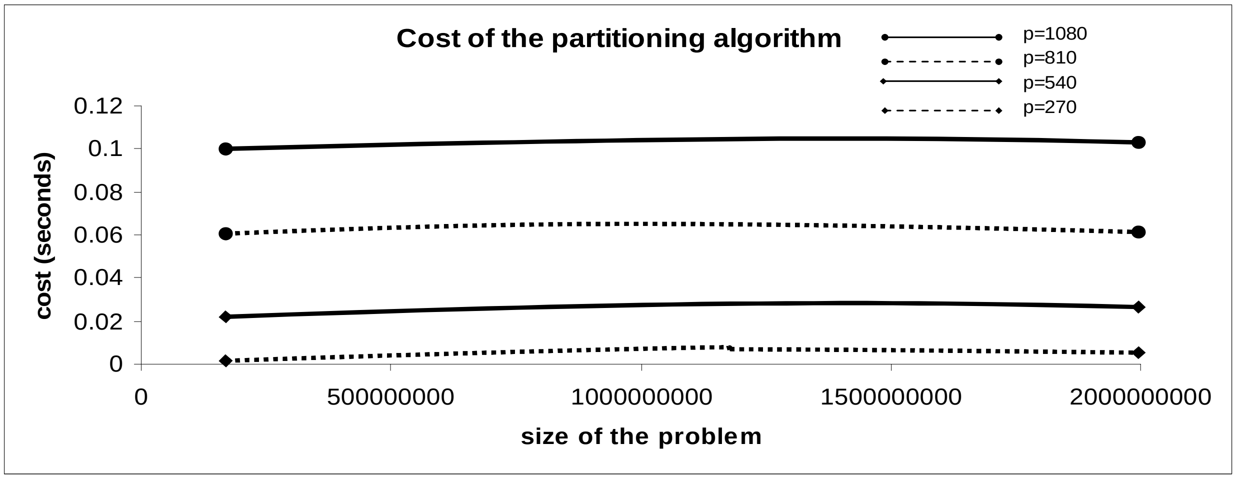


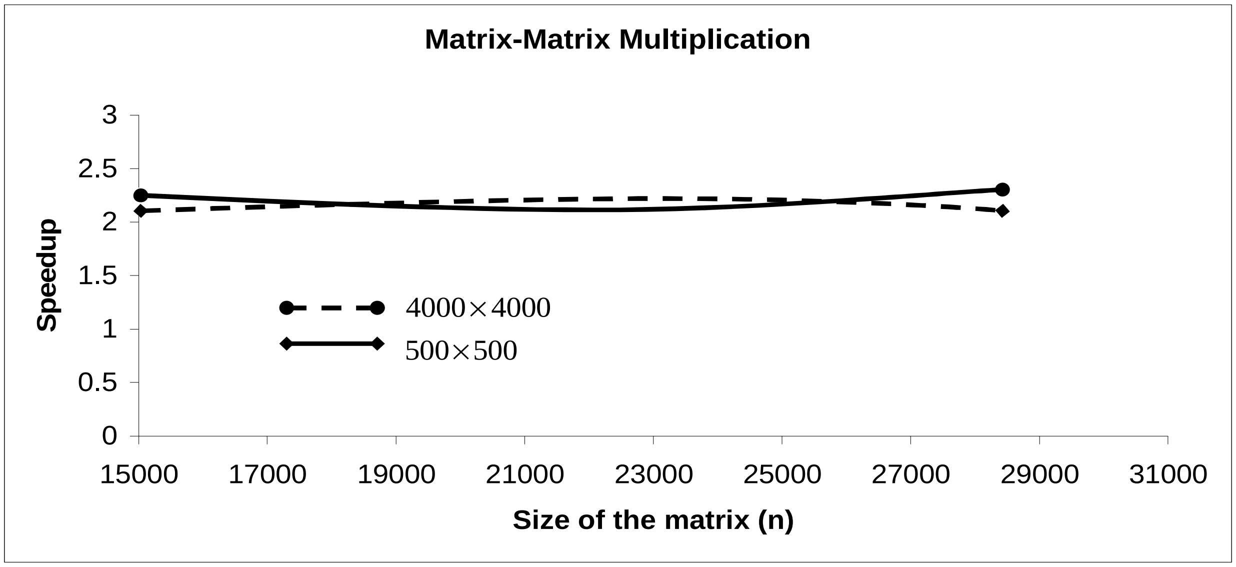
Figure 4.40: The cost of finding the optimal solution using the partitioning algorithm. **p** is the number of processors.

$\frac{130}{19} \approx 6.8$ suggests that the processor set is reasonably heterogeneous and it should also be noted that paging has not started happening at this problem size for both the computers.

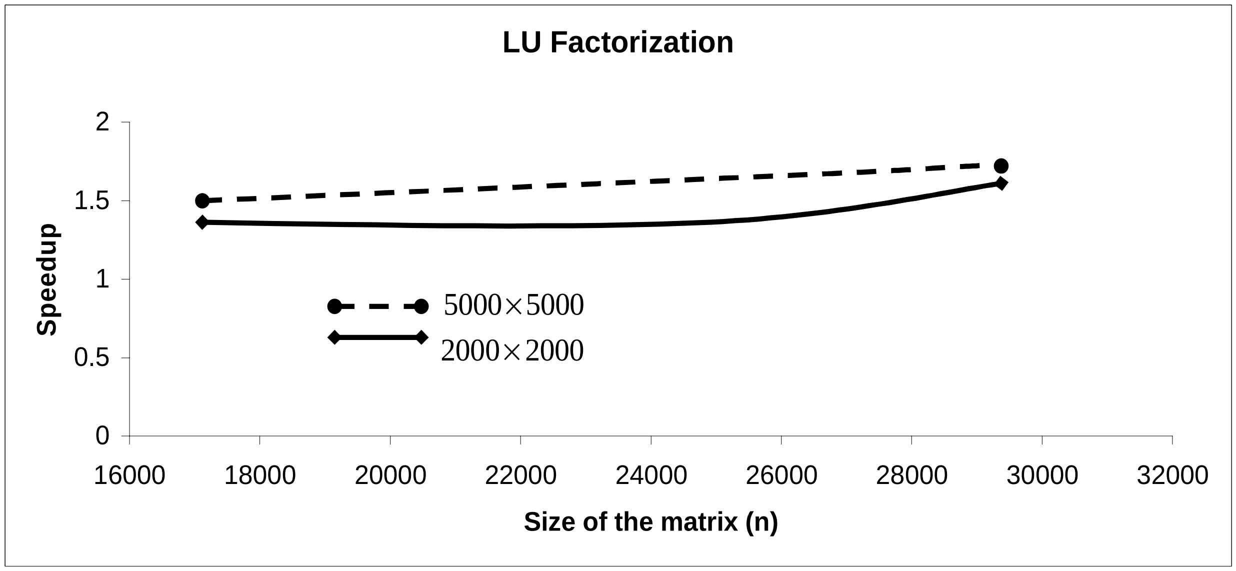
Figure 4.40 displays the cost in seconds of finding the optimal solution using the partitioning algorithm for varying number of processors for large problem sizes. The speed function for each processor is built using the above procedure (5 experimental points appeared enough to build the functions). It can be inferred that this cost is negligible compared to the execution time of the applications which varies from minutes to hours.

4.7.1.1.2 Numerical Results

In this section, we present the experimental results comparing the data partitioning algorithms using the functional model over the data partitioning algorithms using the single number model. In the figures, for each problem size, the speedup calculated is the ratio of the execution time of the application using the single number model over the execution time of the application using the functional model.



(a)



(b)

Figure 4.41: Results obtained using the network of heterogeneous computers shown in Table 4.8. The speedup calculated is the ratio of the execution time of the application using the single number model over the execution time of the application using the functional model. (a) Comparison of speedups of matrix-matrix multiplication. For the single number model, the speeds are obtained using serial matrix-matrix multiplication of two dense square matrices. For the solid lined curve, the matrices used are of size 500×500. For the dashed curve, the matrices used are of size 4000×4000. (b) Comparison of speedups of LU factorization. For the single number model, the speeds are obtained using serial LU factorization of a dense square matrix. For the solid lined curve, the matrix used is of size 2000×2000. For the dashed curve, the matrix used is of size 5000×5000.

For each processor, the piecewise linear function approximation of the real-life speed function and not speed band is built. The piecewise linear function approximation of the real-life speed function for a processor is built using a set of experimentally obtained points (**x**,**s**) . To obtain an experimental point for a problem size **x**, we execute the application for the problem size at that point. The absolute speed of the processor **s** for this problem size is obtained by dividing the total volume of computations by the real execution time (and not the ideal execution time).

The experimental results show that the parallel applications using the functional model demonstrate good speedup over parallel applications using the single number model. At a first glance, it may look strange that there is no problem size where the single number model demonstrates the same speed as the functional model. Actually in heterogeneous environment, the distribution given by the single number model cannot in principle be better than the distribution given by the functional model. This is because the speeds used in the single number model are obtained based on the fact that all the processors get the same number of elements and hence solve problems of the same size as in a homogeneous environment. Consider for example an application employing a serial matrix-matrix multiplication algorithm, the absolute speeds of the processors for this application to be used in the single number model are calculated based on a particular size of matrix, that is, the same number of elements. So whatever problem size is used, it will give wrong estimation of distribution for at least one processor.

Figure 4.41(a) shows the speedup of the matrix-matrix multiplication executed on this network using the functional model over the matrix-matrix multiplication using the single number model. There are two curves, the solid lined curve corresponds to the single number speed of the processor obtained based on the multiplication of two dense 500×500 matrices and the dashed curve corresponds to the single number speed of the processor obtained based on the multiplication of two dense 4000×4000 matrices.

Figure 4.41(b) shows the speedup of the matrix factorization executed on this network using the functional model over the matrix factorization using the single number model. There are two curves, the solid curve corresponds to the single number speed of the processor obtained based on the matrix factorization of a dense 2000×2000 matrix and the dashed curve corresponds to the single number speed of the processor obtained based on the matrix factorization of a dense 5000×5000 matrix.

As can be seen from the figures, the functional model performs better than the single number model for a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds cannot be accurately approximated by constant functions of problem size and our data partitioning algorithms using this model deliver efficient solutions.

4.7.2 Algorithms for Partitioning Sets with Processor Memory Bounds

In previous section, we addressed the problem of optimal distribution of computational tasks on a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds cannot be accurately approximated by constant functions of problem size. We designed efficient algorithms of data partitioning using a realistic performance model of network of heterogeneous computers. This model integrates many essential features of a network of heterogeneous computers having a major impact on its performance such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging. Under this model, the speed of each processor is represented by a continuous and relatively smooth function of the size of the problem whereas standard models use single numbers to represent the speeds of the processors.

We then formulated a problem of partitioning of an **n**-element set over **p** heterogeneous processors using this model and designed efficient algorithms for its solution whose worst-case complexity is $O(\mathbf{p}^2 \times \log_2 \mathbf{n})$ but the best-case complexity is $O(\mathbf{p} \times \log_2 \mathbf{n})$. The optimal solution is the solution where the size of the problem assigned to each processor is proportional to the speed of the processor. The algorithms are based on the following observation: If a distribution of the elements of the set amongst the processors is obtained such that the number of elements is proportional to the speed of the processor, then the points, whose coordinates are number of elements and speed, lie on a straight line passing through the origin of the coordinate system and intersecting the graphs of the processors with speed versus the size of the problem in terms of the number of elements. The algorithms use the observation that the optimal solution obtained by these algorithms is a straight line passing through the origin of the coordinate system and

Machine Name	Architecture	cpu MHz	Total Main Memory (kBytes)	Cache (kBytes)
Comp1	Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM)	2783	7933500	512
Comp2	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	2048
Comp3	Windows XP	3000	1030388	512
Comp4	Linux 2.4.7-10 i686	730	254524	256

Table 4.11: Specifications of the four heterogeneous computers, on which applications are run to determine the effect of caching and paging in reducing their execution speed.

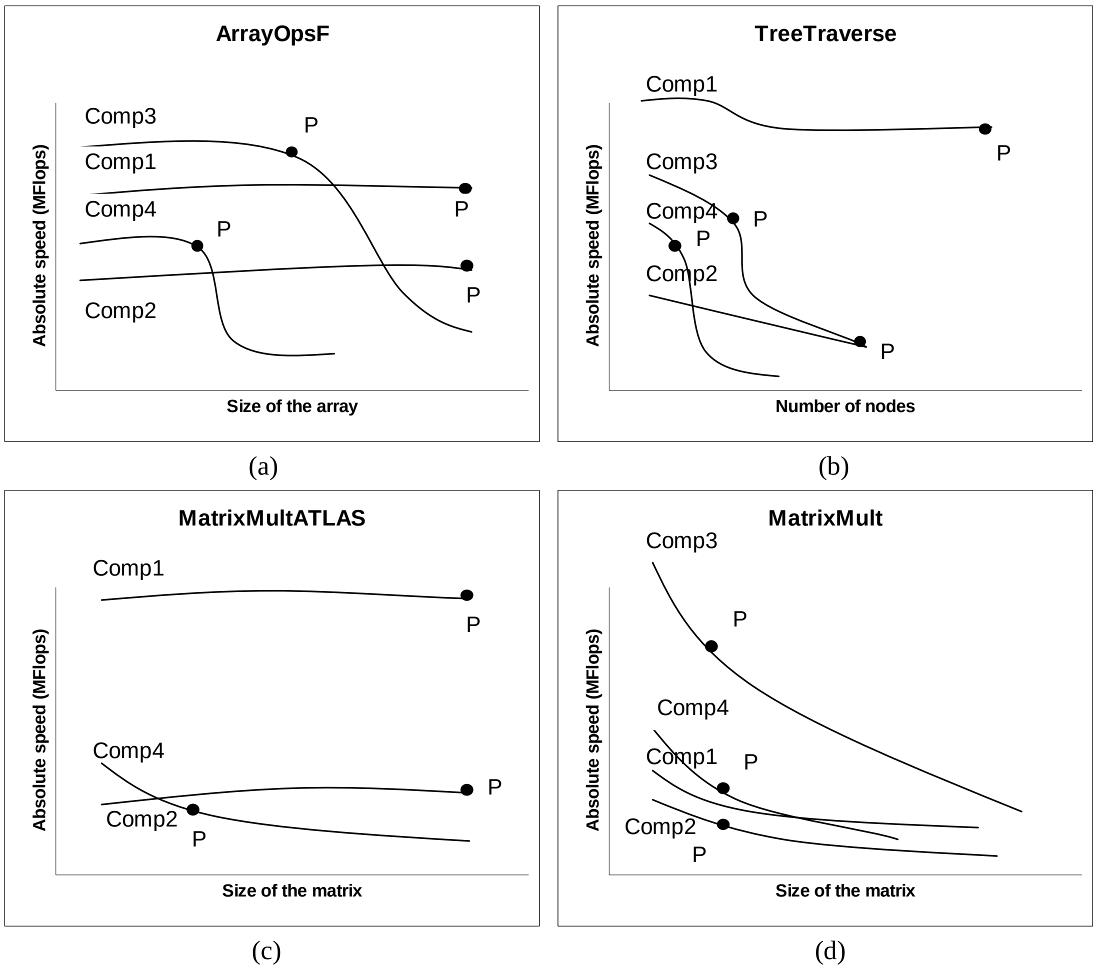


Figure 4.42. The effect of caching and paging in reducing the execution speed of each of the four applications run on network of heterogeneous computers shown in Table 4.11. (a) ArrayOpsF, (b) TreeTraverse, (c) MatrixMultATLAS, and (d) MatrixMult. P is the point where paging starts occurring.

intersecting the graphs of the processors with speed versus the size of the problem in terms of the number of elements. The algorithms take at most $\mathbf{p}^2 \times \log_2 \mathbf{n}$ steps to find the optimal solution.

However this model fails to provide optimal solutions when the network consists of computers that are configured to avoid paging. Consider the experiments shown in Figure 4.42. The experiments show that `Comp1` and `Comp2` do not permit paging. This is typical of computers used as a main server. For applications designed to efficiently use cache memory, such computers show a constant speed function, up to a point where the process crashes, probably because it tries to invoke a paging procedure, not allowed due to its configuration. So if we have such computers, the real speed function of the size of the problem is not continuous any more but discontinuous at the point where paging happens, that is, there is a break in the continuity of the function at the point where paging happens.

Consider a small network of three processors, whose speeds as functions of problem size are shown in Figure 4.43. The processor represented by the speed function $s_1(x)$ is configured to permit paging. The processors represented by speed functions $s_2(x)$ and $s_3(x)$ are configured to avoid paging. The bold curves represent the experimentally obtained parts of the speed functions. Now assume that we want to obtain optimal distributions for problem sizes whose optimal solution lines lie beyond the bold curves. In this case we naturally extrapolate the curves in a continuous manner using some reasonable approximations. The extrapolations are shown by dotted curves. However it can be seen that sometimes the extrapolations are not accurate representations of the real shape of the speed functions as shown for the speed functions $s_1(x)$ and $s_2(x)$. The real speed functions are shown by dashed curves. Consider two data distributions obtained by the functional model and which are shown by dotted lines passing through the origin. Although the first data distribution (x_{11},x_{12},x_{13}) is not the optimal solution just because the

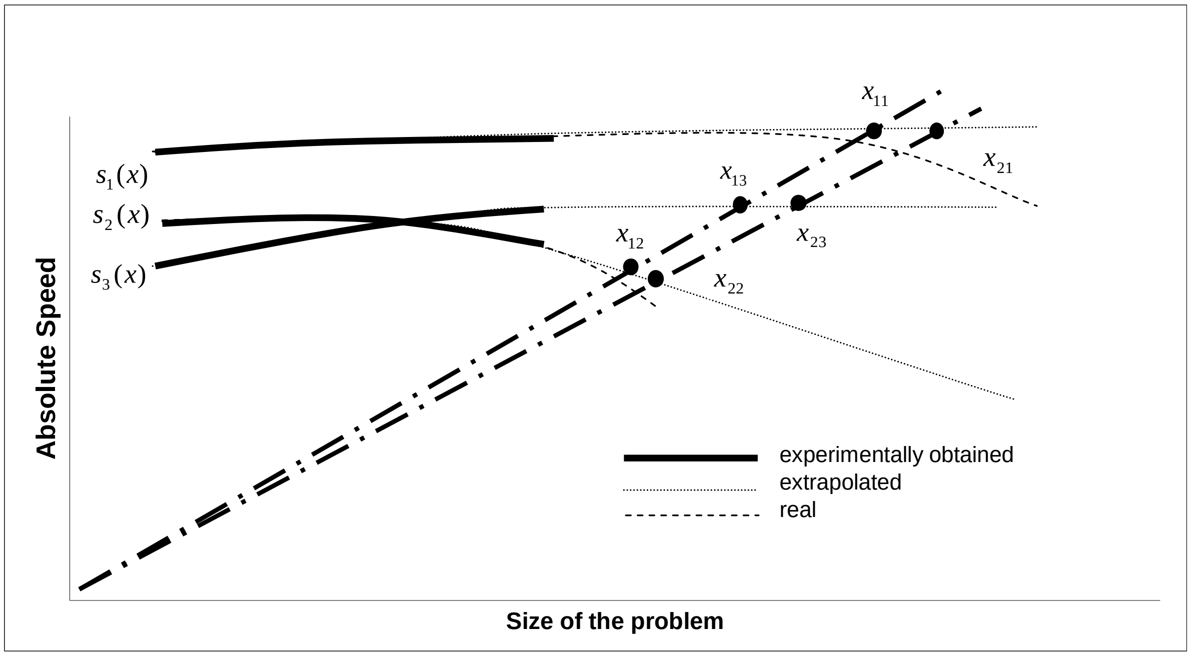


Figure 4.43: A small network of three processors whose speeds are shown against the size of the problem. The dotted lines passing through the origin represent solutions provided by the functional model. The bold curves represent the experimentally obtained speed functions. The dotted curves represent reasonable approximations of the speed functions in a continuous manner. The dashed curves represent the real behavior of the speed functions. The first dotted line giving the data distribution (x_{11},x_{12},x_{13}) is a non-optimal solution. The second dotted line giving the data distribution (x_{21},x_{22},x_{23}) is not a solution at all.

extrapolated speed functions $s_1(x)$ and $s_2(x)$ are not accurate representations of the real speed functions, it still give a reasonable sub-optimal solution of the problem. At the same time, the second data distribution (x_{21},x_{22},x_{23}) is not a solution at all. This is because at the points x_{22} and x_{23} the paging starts occurring for computers with speed functions $s_2(x)$ and $s_3(x)$ and since these computers are configured to avoid paging, they crash. Therefore in order to obtain optimal and working solutions for such networks, we need to extend the functional model.

We naturally extend the functional model by including an additional parameter of maximum problem size. The maximum problem size represents the upper bound on the size of the problem that each processor can solve. For computers that are configured to avoid paging, it represents the point where the computer crashes due to the occurrence of paging and where the speed function of the size of the problem becomes discontinuous.

In the next section, we present the modified functional model. This is followed by a formulation of a general set-partitioning problem, which is the problem of partitioning of an **n**-element set over **p** heterogeneous processors using this modified functional model. Then we give its efficient solution of the complexity $O(\mathbf{p}^3 \times \log_2 \mathbf{n})$. This problem is a simple variant of the most advanced problem of partitioning a set with weighted elements formulated in Section 4.3. We use the simple variant to explain how complex the problem of scheduling tasks amongst processors is when: (a) the processors have significantly different memory structure, and (b) there are memory limitations on the size of task that can be solved by each processor. We also use this variant to explain in simple terms how the modified functional model can be used to achieve better data partitioning on networks of heterogeneous computers before moving on to solve the most advanced problem.

To demonstrate the efficiency of the modified functional model, we perform experiments using naïve parallel algorithms for linear algebra kernel, namely, matrix multiplication and LU factorization using striped partitioning of matrices on a local network of heterogeneous computers. Our main aim is not to show how matrices can be efficiently multiplied or efficiently factorized but to explain in simple terms how the modified functional model can be used to optimally schedule tasks on networks of heterogeneous computers taking into account the processor and memory heterogeneity. We also view these algorithms as good representatives of a large class of data parallel computational problems and a good testing platform before experimenting more challenging computational problems.

We end the discussion with a survey of related literature.

4.7.2.1 The Extended Performance Model of Networks of Heterogeneous Computers

The modified functional model of networks of heterogeneous computers has the following parameters:

- An upper bound on the size of the task that can be solved by each computer, and
- The speed of the processor is represented by a continuous and smooth function of the problem size until the upper bound. Beyond the upper bound, the speed of the processor is assumed to be zero.

The model retains the restrictions imposed by the functional model on the shape of the graph representing the speed function. The shape of the graph should be such that there is only one intersection point of the graph with any straight line passing through the origin. That is the speeds of the processors must either be increasing or decreasing functions of problem size for the problem sizes for which the solutions are sought. These assumptions on the shapes of the graph are representative of the most general shape of graphs observed for applications experimentally as shown in Figure 4.42.

The upper bound could signify one of the following cases:

- Allocation of a task whose size is beyond this bound could result in processor failure.
- Allocation of the task whose size is beyond this bound could result in unacceptable execution time to accomplish the task due to severe paging.

4.7.2.2 Algorithm for Partitioning a Set with Processor Memory Bounds

Using the modified functional model, we solve the following problem of partitioning a set, which can be formulated as:

Definition 1. *Heterogeneous Memory Partitioning HMP(n, s, b):*

Given: (1) A set of **n** elements, and (2) A well-ordered set of **p** heterogeneous processors whose speeds are functions of the size of the problem **x**, $s_i = f_i(x)$, and (3) There is a upper bound on the largest problem size that can be solved on each processor, that is, there is an upper bound b_i on the number of elements stored by each processor (i=0,…,**p**-1);

Partition the set into **p** disjoint partitions such that:

- $x_0+x_1+...+x_{p-1}=n$, where $x_0,x_1,...,x_{p-1}$ are the number of elements in partitions 0,1,…,**p**-1 respectively,
- $x_i \le b_i$ for all (i=0,…,**p**-1),
- the maximum $\max_{i=0}^{p-1}(\frac{x_i}{s_i})$ of the execution times of the processors is minimized. That is solve the following min-max problem:

$$\min\left\{\max_{i=0}^{p-1}(\frac{x_i}{s_i})\right\}$$

where x_i is the number of elements in partition **i**. We assume that the volume of computations involved in the execution of a problem size is proportional to the problem size.

We provide an optimal solution to this problem of complexity O($p^3 \times \log_2 n$).

When there is an upper bound b_i on the number of elements stored by each processor (**i**=0,…,**p**-1), the algorithm used to solve the partitioning problem is of complexity O(p^3). This algorithm can be summarized as follows:

1. Partition the set such that the number of elements in each partition is proportional to the speed of the processor and assuming no upper bound exists on the number of elements that can be stored by each processor. If the number of elements assigned to each processor is less than or equal to the upper bound on the number of elements that can be stored by each processor, we have the optimal distribution.
2. For each processor $\mathbf{i}$ ($i=0,\ldots,\mathbf{p}-1$), we check if the number of elements assigned to it is greater than the upper bound on the number of elements that it can store. For all the processors whose upper bounds are exceeded, we assign them the number of elements equal to their upper bounds. Now we solve the partitioning problem of a set with remaining elements over the remaining processors. We recursively apply this procedure until all the elements have been assigned.

The proof of optimality of the solution provided by this algorithm is given in [WS04]. This is indeed a special case of the problem variant we are going to solve in this section.

When the speed of the processor is represented by a function of the size of the problem, $\mathbf{s=f(x)}$, and when there is no upper bound on the number of elements stored by each processor, efficient algorithms of complexity $O(\mathbf{p}^2\times\log_2\mathbf{n})$ have been presented in the previous section.

When the speed of the processor is represented by a function of the size of the problem, $\mathbf{s=f(x)}$, and when there is an upper bound on the number of elements stored by each processor, the problem of partitioning a set is non-trivial. Before presenting the algorithm to solve this problem, we formulate the formal mathematical problem of the optimization problem HMP of partitioning of the set. Given: (1) A set of $\mathbf{n}$ elements, and (2) A well-ordered set of $\mathbf{p}$ functions, $\mathbf{s_i=f_i(x)}$, and (3) There is a upper bound $\mathbf{b_i}$ on the number of elements that can be stored in each partition ($i=0,\ldots,\mathbf{p}-1$), find a partition of the set into $\mathbf{p}$ disjoint partitions such that:

- $x_0+x_1+...+x_{\mathbf{p}-1}=\mathbf{n}$ where $x_0,x_1,...,x_{\mathbf{p}-1}$ are the number of elements in partitions $0,1,\ldots,\mathbf{p}-1$ respectively,
- $x_i \le b_i$ for all $(i=0,\ldots,\mathbf{p}-1)$,
- the maximum of $\max_{i=0}^{p-1}(\frac{x_i}{s_i})$ is minimized. That is solve the following min-max problem:

$$\min\left\{\max_{i=0}^{p-1}(\frac{x_i}{s_i})\right\}$$

where x_i is the number of elements in partition **i**.

Before we present the algorithm to solve the optimization problem HMP, we apply the following assumptions:

(1) The speed of each processor is represented by a continuous function of the size of the problem up till its upper bound on the problem size. The speed of the processor is zero beyond the upper bound.

(2) The shape of the graph representing the speed function should be such that there is only one intersection point of the graph with any straight line passing through the origin. That is the speeds of the processors must either be increasing or decreasing functions of problem size for the problem sizes for which the solutions are sought and,

(3) For each processor, for all $x \ge y$, where x and y are problem sizes, the execution times t_x and t_y to execute problems of sizes x and y respectively are related by $t_x \ge t_y$.

Algorithm *Heterogeneous Memory Partitioning Algorithm HMPA(n, s, b).* The algorithm we propose to solve this advanced partitioning problem is graphically illustrated in Figure 4.44 and has the following main points:

1. Partition the set such that the number of elements in each partition is proportional to the speed of the processor and assuming no upper bound exists on the number of elements that

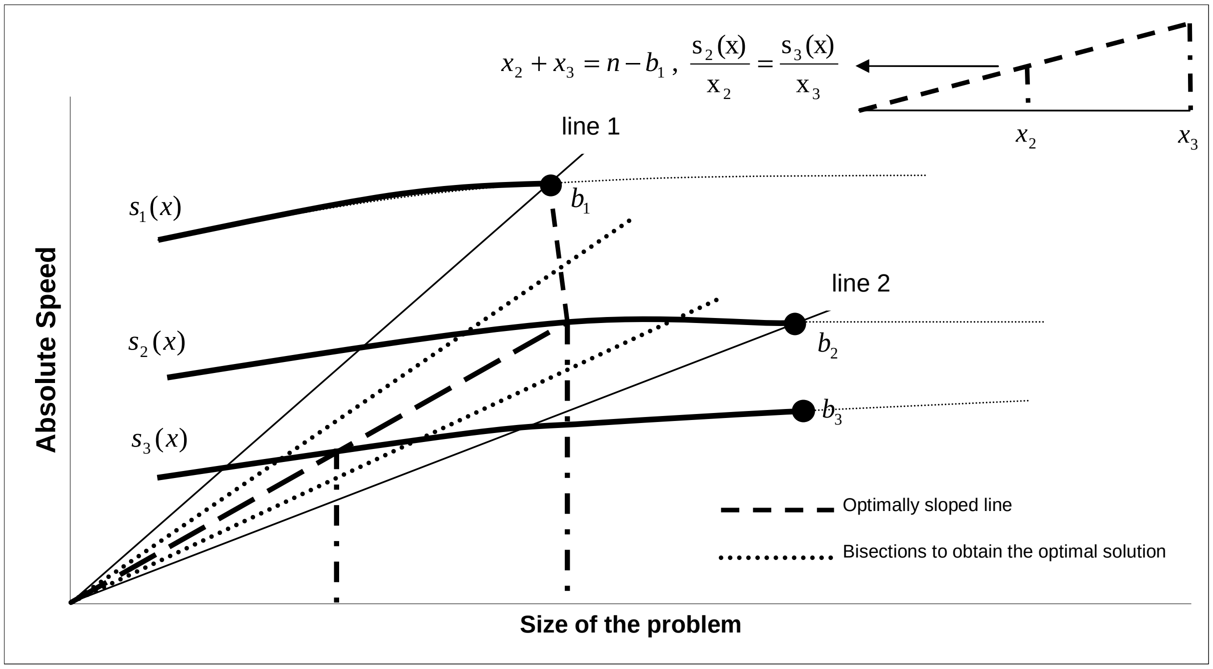


Figure 4.44: The partitioning algorithm for the problem size **n**. The bold curves represent the experimentally obtained speed functions. The dotted curves represent reasonable approximations of the speed functions in a continuous manner. For processor represented by speed function $s_1(x)$, we assign this processor the number of elements equal to its upper bound $\mathbf{b_1}$. We then partition the set with remaining $\mathbf{n\text{-}b_1}$ elements amongst the processors represented by speed functions $s_2(x)$ and $s_3(x)$ respectively. The region between the lines **line1** and **line2** is bisected to narrow down to the optimal solution.

can be stored by the processor (we can use any continuous extension of the speed function beyond the maximal problem size, say, a constant equal to the speed for the maximal problem size). The partitioning algorithm used to perform this task is discussed in the previous section. If the number of elements in each partition assigned to each processor is less than the upper bound on the number of elements that can be stored by the processor, we have an optimal distribution.

2. For each processor **i** (i=0,…,**p**-1), we check if the number of elements assigned to it is greater than the upper bound on the number of elements that it can store. For all the

processors whose upper bounds are exceeded, we assign them the number of elements equal to their upper bounds. Now we solve the partitioning problem of a set with remaining elements over the remaining processors. We recursively apply this procedure until all the elements have been assigned.

Theorem 1. *HMPA(n, s, b) gives the optimal solution to the optimization problem HMP(n, s, b).*

Proof. We prove the optimality of the solution using mathematical induction. We use the maximum time to solve the task assigned to each processor as the performance metric.

The cases for **p**=1 and **p**=2 are trivial. For **p**=3, let us assume the upper bounds of the processors 1, 2, and 3 on the number of elements that they can store are b_1, b_2, and b_3 respectively. Suppose the optimal distribution assuming there are no upper bounds on the number of elements is (x_1, x_2, x_3) such that x_1+x_2+x_3=**n** where **n** is the size of the problem.

Consider the case where $x_1 > b_1$ and $x_2 > b_2$. Let us assign the number of elements equal to b_1 for processor 1. The remaining distribution has to satisfy the equality $x_2^{'} + x_3^{'} = n - b_1$ where $x_2^{'}$ and $x_3^{'}$ are to be chosen such that the speed of the processor is proportional to the number of elements assigned to it. If the speeds of the processors 2 and 3 are non-increasing functions of problem size, it can be proved that $x_2^{'} > x_2$ and $x_3^{'} > x_3$. This gives us the inequality $x_2^{'} > x_2 > b_2$. Therefore we have to necessarily assign b_2 number of elements to processor 2. If the speeds of the processors 2 and 3 are non-decreasing functions of problem size, there are three possibilities, ($x_2^{'} > x_2, x_3^{'} > x_3$), ($x_2^{'} < x_2, x_3^{'} > x_3$) and ($x_2^{'} > x_2, x_3^{'} < x_3$). The first and the third possibility give us the inequality $x_2^{'} > x_2 > b_2$. For the second possibility, any allocation $x_2^{''}$ such that $x_2^{''} < b_2$ would result in an allocation of $x_3^{''}$ number of elements to processor 3 such that $x_3^{''} > x_3^{'}$ thus resulting in a larger execution time. Therefore we have to necessarily assign b_2

number of elements to processor 2. If the speed of the processor 2 is a non-decreasing function of problem size and speed of processor 3 is a non-increasing function of problem size, there are two possibilities, ($x_2^{'} < x_2, x_3^{'} > x_3$) and ($x_2^{'} > x_2, x_3^{'} < x_3$). In the first possibility, any allocation $x_2^{''}$ such that $x_2^{''} <$ b2 would result in an allocation of $x_3^{''}$ number of elements to processor 3 such that $x_3^{''} > x_3^{'}$ thus resulting in a larger execution time. The second possibility gives us the inequality $x_2^{'} > x_2 > b_2$. Therefore we have to necessarily assign b_2 number of elements to processor 2.

Consider the case of optimal distribution where $x_1 > b_1$ is true. For processor 1, we assign the number of elements equal to b_1. The remaining elements are allocated such that $x_2^{'} + x_3^{'} = n - b_1$ where $x_2^{'}$ and $x_3^{'}$ are to be chosen such that the speed of the processor is proportional to the number of elements assigned to it. Any other allocation $x_1^{''}$ such that $x_1^{''} <$ b_1 would result in an allocation where one of the inequalities ($x_2^{''} > x_2^{'}$), ($x_3^{''} > x_3^{'}$) is satisfied thus resulting in a larger execution time. It can be proved similarly for the case when $x_2 > b_2$.

Assuming this to be true for **p**=**k** processors, we have to prove the optimality for **p**=**k**+1 processors. For a given problem size **n**, let us assume the distribution given by our algorithm to be $x_0, b_1, b_2, \cdots, b_m, x_{m+1}, \cdots, x_k$ such that $x_0 + b_1 + \cdots + x_k = n$, where without loss of generality processors 1,…,**m** are allocated their upper bounds. It can be inferred that the execution times for the rest of the processors 0,**m**+1,…,**k** satisfy the equality $t_0 = t_{m+1} = \cdots = t_k$. It can also be inferred that $(t_0, t_{m+1}, \cdots, t_k) \geq t_i$ for all i=1,…,**m**. The execution time for the problem size is equal to $t_{mp} = \max_{i=0}^{k}(t_i) = (t_0, t_{m+1}, \cdots, t_k)$. Consider an alternative solution with the distribution $x_0^{'}, x_1^{'}, \cdots, x_k^{'}$ where $x_0^{'} + x_1^{'} + \cdots + x_k^{'} = n$ and $x_1^{'} \leq b_1, \cdots, x_m^{'} \leq b_m$. It can be easily seen that for atleast one processor i (i=0,**m**+1,…,**k**), $x_i^{'} \geq x_i$, thus giving an execution time $t_i^{'}$, which is greater than the execution time given by our algorithm t_{mp}.

Theorem 2. *The complexity of the algorithm HMPA(n, s, b) is O(**p**^{3}×log$_2$**n**).*

Proof. There are **p** major steps in the algorithm. At each such major step **i**, we solve the problem of partitioning of a set amongst **p-i** processors such that the number of elements in each partition is proportional to the speed of the processor and assuming no upper bound exists on the number of elements that can be stored by the processor. The complexity of this step is O(**p**^{2}×log$_2$**n**) as discussed in the previous section. Since there are **p** such steps, the overall worst-case complexity is O(**p**^{3}×log$_2$**n**). Mathematically, the worst-case complexity is the summation of **p** terms:

$$\begin{aligned} C &= p^2 \times \log_2 n + (p-1)^2 \times \log_2 (n-b_0) + (p-2)^2 \times \log_2 (n-b_0-b_1) + \cdots + 1 \\ &\cong p^2 \times \left(\log_2 n + \log_2 (n-b_0) + \log_2 (n-b_0-b_1) + \cdots\right) \\ &\cong p^2 \times \left(\log_2 (n \times (n-b_0) \times (n-b_0-b_1) \times \cdots)\right) \\ &\cong p^2 \times (\log_2 n^p) \\ &\cong p^3 \times \log_2 n \end{aligned}$$

4.7.2.3 Applications of the Model

So far we have formulated a realistic performance model of a network of heterogeneous computers and designed efficient algorithms of data partitioning with this model. Now we present a list of practical applications of this model:

- Data partitioning on networks of heterogeneous computers, which only include computers that are configured to avoid paging. Such computers crash when problem sizes are allocated that requires paging. The largest problem size on such computers is the problem size where paging starts happening.
- Data partitioning on networks of heterogeneous computers, which only include computers that permit paging. However allocation of large problem sizes can cause severe paging on such computers as a result causing severe performance degradation and sometimes stalling

of the entire application. The largest problem size on such computers is not the problem size where paging starts happening but the problem size which causes severe performance degradation of the application.

- Data partitioning on networks of heterogeneous computers, which include computers some of which permit paging and some of which are configured to avoid paging.

4.7.2.4 Experimental Results

The experimental results are divided into three sections. The first two sections are devoted to building the modified functional model. In the first section, we suggest ways to determine the upper bound on the size of the problem that each processor can solve. Then we present the parallel applications and the network of heterogeneous computers on which the applications are tested. For each application, we explain how to estimate the processor speed. This is followed by presentation of the procedure to build the speed functions of the processors. Finally we present the experimental results obtained by running these applications on the network of heterogeneous computers.

4.7.2.4.1 Determination of Largest Problem Size

In this section, we highlight different approaches to determine the largest problem size of an application that can be solved efficiently on a given computer. We do not define the notion of largest problem size as this depends on the nature of the applications run on the network of heterogeneous computers and the level of integration of the computers in this network.

One of the ways is to determine the user-available memory on the computer and the memory requirement of the application. If the memory requirement of the application is less than the user-available memory then the application will not suffer from memory limitations. We can

```
shell$ cat /proc/meminfo
MemTotal:      1033908 kB
MemFree:        389568 kB
…
shell$ top
Mem:  1033908k total,  644340k used,   389568k free,   512680k buffers
Swap: 2040212k total,    7924k used,  2032288k free,    36916k cached
…
```

Figure 4.45: Operating system tools to determine the user-available memory for an application. The user-available memory is highlighted in bold.

determine the largest problem size we can run, by calculating when the total memory requirement of an application would exceed the user-available memory capacity on a given computer. The total user-available memory of a computer can be obtained from the operating system utilities like ‘**`cat /proc/meminfo`**’ and ‘**`top`**’ as shown in Figure 4.45. There are also system calls that can be called from the application code to obtain the user-available memory of a given computer.

Cierniak *et al.* [CLZ97] show that the total memory requirement is generally not a good criterion for judging the largest problem size that can be run efficiently. The reason is that the total memory requirement is a very conservative measure, and generally overestimates the memory requirement of an application. They introduce a new notion, the resident memory size (RMS) for a given program segment, defined as the minimum number of pages of physical memory required to ensure that all fault misses are cold misses (i.e. due to the first reference) for that segment, using a particular page replacement algorithm. If the resident memory size is less than the user-available memory then the application will not suffer from the effects of memory limitations. If, on the other hand, the program’s RMS is larger than the available memory then some of the pages required will not be in memory, and a page fault occurs. As the input data size increases, the RMS increases, ultimately exceeding the available memory. A compile-time algorithm is provided to approximate the RMS. The notion of RMS value should work well in practice for regular problems, but it may not be a good approximation for irregular problems.

As shown in Figure 4.42, the notion of the largest problem size depends on the nature of the application and on the level of the integration of the computers used in the experiments. For computers that do not permit paging, the largest problem size is the point where paging starts happening. This is shown to be the point P for computers `Comp1` and `Comp2` in Figure 4.42 for all the applications. For computers configured to permit paging, the largest problem size is not the point where paging starts happening but the point where the absolute speed of the processor falls drastically. This is shown to be the point P for computers `Comp3` and `Comp4` in Figure 4.42 for all the applications.

The problem size at point P shown in Figure 4.42 is probably less than the largest problem size but it is a good approximation. Speed functions built with large number of points with a wider range of problem sizes can give a better approximation of largest problem size that can be solved on a processor. However in this case it depends on a number of conditions such as how much time the application programmers are willing to spend to build the speed functions of the processors and their level of efficiency. This approach of determining the largest problem size should work well in practice for regular as well as irregular problems.

We aim to perform more future work to determine accurately the problem size at which paging starts happening for both regular as well as irregular problems. We aim to do this determination with minimal experimental time.

Machine Name	Architecture	cpu MHz	Total Main Memory (kBytes)	Largest size of task (MM)	Largest size of task (LU)	Cache (kBytes)
X1	Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM)	2783	7933500	116640000	262440000	512
X2	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	36000000	81000000	512
X3	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	36000000	81000000	512
X4	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	36000000	81000000	512
X5	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1030508	36000000	81000000	512
X6	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	31360000	64000000	2048
X7	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	30250000	59290000	2048
X8	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	30250000	64000000	2048
X9	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	524288	30250000	59290000	2048
X10	Linux 2.4.18-3 i686 Intel Pentium III	997	254576	24502500	30250000	256
X11	SunOS 5.5 Sun4m sparc SUNW,SPARCstation-5	110	65536	6000000	6250000	512

Table 4.12: Specifications of the eleven heterogeneous processors to demonstrate the efficiency of the modified functional model.

4.7.2.4.2 Applications

A small heterogeneous local network of 11 different Solaris and Linux workstations shown in Table 4.12 is used in the experiments. The network is based on 100 Mbit Ethernet with a switch enabling parallel communications between the computers.

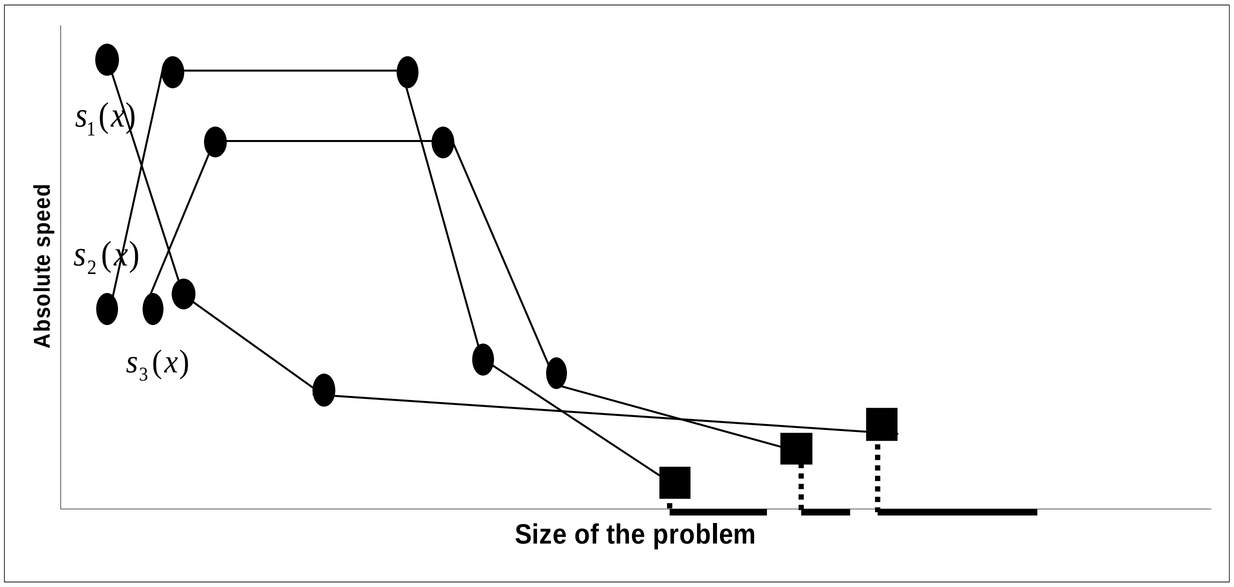


Figure 4.46: Using piece-wise linear approximation to build speed functions for 3 processors. The circular points are experimentally obtained whereas the square points represent the upper bounds. The speed function for processor $s_1(x)$ is built from 3 experimentally obtained points (application run on this processor uses memory hierarchy inefficiently) whereas the speed functions $s_2(x)$ and $s_3(x)$ (application run on these processors use memory hierarchy efficiently) are built from 4 experimentally obtained points. Speeds of the processors are assumed to be zero for problem sizes beyond their upper bounds.

The two applications used to demonstrate the efficiency of the modified functional model over the functional and the single number models are described in detail in Section 4.7.1.1.1. The procedure to build the piece-wise linear function approximation is also described in detail. We use piece-wise linear function approximation illustrated in Figure 4.46 to build the speed function. Such approximation of the speed function is compliant with the requirements of the model, which are the shape requirements of the graph representing the speed function and that the speeds be continuous and smooth functions of problem size up till its upper bound on the problem size and zero beyond.

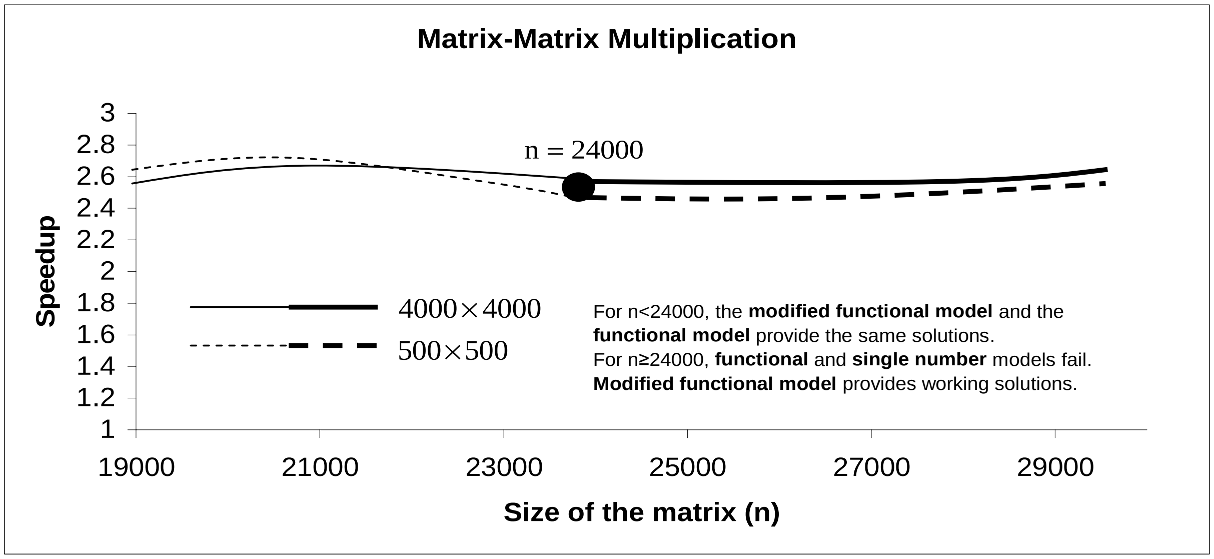
4.7.2.4.3 Numerical Results

In this section, we present the experimental results demonstrating the efficiency of our modified functional model over the functional and the single number models.

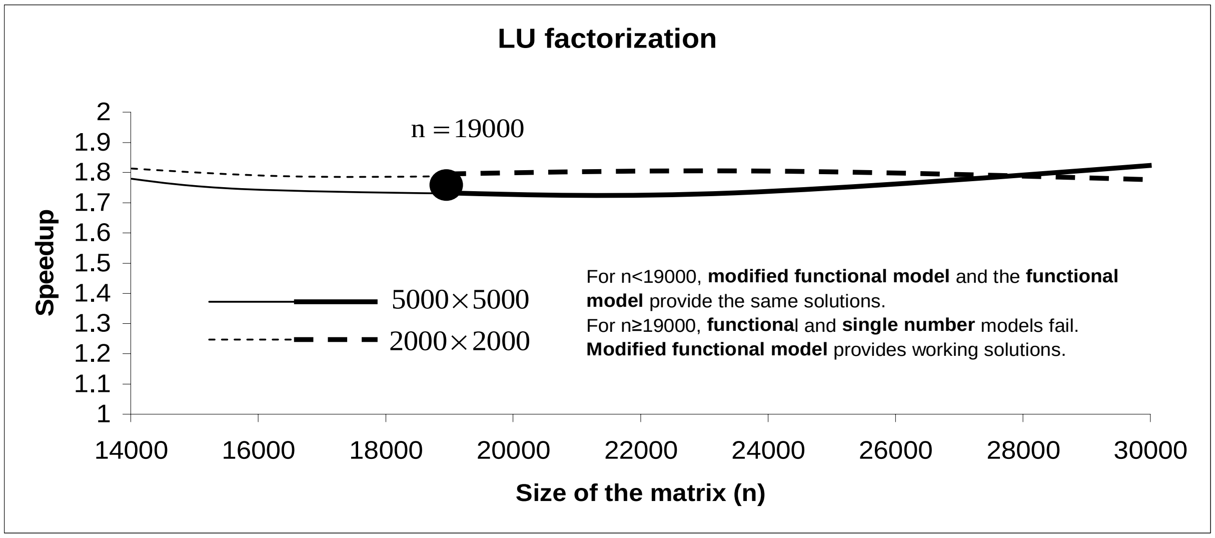
In the figures, the speedup calculated is the ratio of the execution time of the application using a single number model over the execution time of the application using a functional model. A set of as few as 5 points is used to build the speed functions of the processors for the functional models.

The solid lined and dashed curves with normal thickness represent the speedup obtained using the functional model over the single number model [BBP+01]. Both these models do not take into account the upper bounds on the problem size that a processor can solve. The solid lined and dashed curves with bold thickness represent the speedup obtained using the modified functional model over the single number model [WS04]. Both these models take into account the upper bounds on the problem size that a processor can solve.

Figure 4.47(a) shows the speedup of the matrix-matrix multiplication executed on this network using the functional models over the matrix-matrix multiplication using the single number model. There are two curves, the solid lined curve corresponds to the single number speed of the processor obtained based on the multiplication of two dense 500×500 matrices and the dashed curve corresponds to the single number speed of the processor obtained based on the multiplication of two dense 4000×4000 matrices. It can be seen from the figure that problem sizes beyond 24000 cannot be solved by using the functional and the single number models. This is because both these models do not take into account the memory limitations of the computers involved in the execution of the application. The modified functional model is used to obtain solutions for problem sizes beyond 24000. It should also be noted that the modified functional



(a)



(b)

Figure 4.47: Results obtained using the network of heterogeneous computers shown in Table 4.12. The speedup calculated is the ratio of the execution time of the application using a single number model over the execution time of the application using a functional model. (a) Comparison of speedups of matrix-matrix multiplication. For the single number models, the speeds are obtained using serial matrix-matrix multiplication of two dense square matrices. For the solid lined curves, the matrices used are of size 4000×4000. For the dashed curves, the matrices used are of size 500×500. (b) Comparison of speedups of LU factorization. For the single number models, the speeds are obtained using serial LU factorization of a dense square matrix. For the solid lined curves, the matrix used is of size 5000×5000. For the dashed curves, the matrix used is of size 2000×2000.

model and the functional model provide the same solutions for problem sizes less than 24000. This is because the data distributions for problem sizes less than 24000 do not exceed the upper bound for any processor. Thus it can be seen that larger problem sizes are solved using modified functional model and the execution performance obtained is good.

Figure 4.47(b) shows the speedup of the matrix factorization executed on this network using the functional models over the matrix factorization using the single number model. There are two curves, the solid lined curve corresponds to the single number speed of the processor obtained based on the matrix factorization of a dense 2000×2000 matrix and the dashed curve corresponds to the single number speed of the processor obtained based on the matrix factorization of a dense 5000×5000 matrix. It can be seen from the figure that problem sizes beyond 19000 cannot be solved by using the functional model and single number models. This is because both these models do not take into account the memory limitations of the computers involved in the execution of the application. The modified functional model is used to obtain solutions for problem sizes beyond 19000. It should also be noted that the modified functional model and the functional model obtain the same solutions for problem sizes less than 19000. This is because the data distributions for problem sizes less than 19000 do not exceed the upper bound for any processor. Thus it can be seen that larger problem sizes are solved using the modified functional model and the execution performance obtained is good.

As can be seen from the figures, the modified functional model performs better than the currently existing models for a network of heterogeneous computers.

4.7.2.5 Related Work

We survey related work in this section. They fall into two categories: papers dealing with task partition and scheduling with memory constraints on dedicated environments and papers dealing with task scheduling with memory constraints on non-dedicated computing environments like the Heterogeneous Networks of Computers (HNOCs) and computing grids.

Li, Bharadwaj, and Ko [LVK00] investigate the problem of scheduling a divisible load onto a set of processors with finite-size buffers in heterogeneous single-level tree networks. They propose a fast algorithm called Incremental Balancing Strategy (IBS) to achieve the optimal processing time. In each increment, distribution of the load is found for processors with available memory according to the standard divisible load theory methods [BGM+96] without taking the memory constraints into account. Then, the distribution of the load is scaled proportionately such that at least one buffer is filled completely. The remaining available buffer capacities are memory sizes in the next increment. This process is continued until distributing the entire load. Drozdowski and Wolniewicz [DW03a] propose a linear programming method of finding solutions with guaranteed optimality for the problem of scheduling divisible loads in networks of processors with limited memory and communication startup times. The complexity of the linear programming solutions that they use to solve their problem is $O(\mathbf{p}^{3.5}\times L)$, where **p** is the number of processors involved in the execution of the algorithm and *L* is the length of the string encoding all the parameters of linear program.

The works discussed take into account the processor heterogeneity in terms of speeds, memory heterogeneity in terms of memory limitation at each processor, and network heterogeneity in terms of the communication cost between a pair of processors. However, these works assume distributed systems with a flat memory model and are not applicable to systems with memory hierarchy. The dependence of the speed of the processor on the size of the problem is assumed to be linear as is usually observed on dedicated distributed multiprocessor computer systems. The largest problem size that can be solved at each processor is assumed to be the core memory at that processor. This is a safe assumption on dedicated distributed multiprocessor computer systems. However on networks of heterogeneous computers, due to the nature of applications run and the level of integration of the computers involved in execution of these applications, the core memory at each processor is just an upper bound on the largest problem size that can be solved but is not a good approximation of the actual largest problem size that can be solved.

The modified functional model that we propose integrates the essential features underlying applications run on a network of heterogeneous computers, mainly, the processor heterogeneity, the heterogeneity of memory structure, and the memory limitations at each level of memory hierarchy. We also present efficient algorithms of data partitioning with this model with relatively low complexity of $O(\mathbf{p}^3 \times \log_2 \mathbf{n})$. However we do not consider the cost of communications in our modified functional model.

While resource management and task scheduling are identified challenges of Grid computing, current Grid scheduling systems mainly focus on CPU and network availability. Many heuristic scheduling algorithms [BWC+03, SW03] have been proposed for traditional high performance computing. However these scheduling systems are for dedicated multiprocessor computer systems and also ignore the impact of memory resource availability on the scheduling decision-making.

Several studies have been reported on task allocation for load balance considering memory resource constraints. An opportunity cost approach proposed in [AAB+00] converts the usage of resources including CPU and memory to a single homogeneous cost. Based on the cost, task is assigned or reassigned to each node for load balance. Load sharing policies with the consideration of effective usage of global memory were studied in [XCZ02]. They consider two types of application workload, known memory demands and unknown memory demands. However their major concern is how to reduce the average slowdown of all individual jobs in the system, instead of how to schedule a parallel application to achieve its best performance. Wu and Sun [WS04] consider how to partition a Grid application and schedule it on a cluster of distributed heterogeneous resources to obtain a minimum application execution time with the consideration of both CPU resource availability and memory resource availability. Three task partition policies, namely, CPU-based, memory-based, and CPU-memory combined partition are studied. They show that the CPU-memory combined approach shows good performance gains over the other approaches. A heuristic CPU-memory algorithm for task scheduling of a meta-task is also proposed. The effect of local jobs on a grid application execution in the situation of resource sharing is evaluated using distribution functions. Currently our modified functional model and the algorithms using this model are not applicable for task scheduling of a meta-task.

The accurate modeling of the electronic structure of atoms and molecules involves computationally intensive tensor contractions involving large multidimensional arrays. The efficient computation of complex tensor contractions usually requires the generation of temporary intermediate arrays. These intermediates could be extremely large, but they can often be generated and used in batches through appropriate loop fusion transformations. To optimize the performance of such computations on parallel computers, Cociorva *et al*. [CBL+02] present a framework to address the optimization problem: given a set of computations expressed as a sequence of tensor contractions, an empirically derived measure of the communication cost for a given target computer, and a specified limit on the amount of available memory on each processor, re-structure the computation so as to minimize the total execution time while staying within the available memory. The framework considers only the heterogeneity in terms of the memory limitations of each computer and is not applicable for programming applications on networks of heterogeneous computers, which exhibits processor heterogeneity in terms of speeds and memory heterogeneity in terms of memory hierarchy and memory limitations of each computer.

4.8 Summary

We have presented a classification of partitioning problems on networks of heterogeneous computers. Our approach to classification of partitioning problems is based on two corner stones:

- A realistic performance model of networks of heterogeneous computers,
- A natural classification of mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) solving problems on networks of heterogeneous computers.

We have proposed a realistic performance model of a network of heterogeneous computers and designed efficient algorithms of data partitioning with this model. This model integrates many essential features of a network of heterogeneous computers having a major impact on its performance such as the processor heterogeneity, the heterogeneity of memory structure, and the effects of paging. Under this model, the speed of each processor is represented by a continuous and relatively smooth function of the size of the problem whereas standard models use single numbers to represent the speeds of the processors.

We designed efficient algorithms of data partitioning using this functional model of network of heterogeneous computers. We particularly addressed the problem of optimal distribution of computational tasks on a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds cannot be accurately approximated by constant functions of problem size.

We then proposed a modified functional model of a network of heterogeneous computers that takes into account the processor heterogeneity, the heterogeneity of memory structure, and the memory limitations at each level of memory hierarchy of a processor. We then designed efficient algorithms of data partitioning with this model thus addressing the problem of optimal distribution of computations over heterogeneous computers taking into account the processor heterogeneity, the heterogeneity of memory structure, and the memory limitations at each level of memory hierarchy of a processor.

The modified functional model proposed can be used to design efficient algorithms of data partitioning for mathematical structures other than sets such as matrices, graphs, and trees. This model can be used to design efficient algorithms for the most general partitioning problem, which can be formulated as:

- Given: (1) An application of problem size $\mathbf{n}$ to be solved, and (2) A well-ordered set of $\mathbf{p}$ processors whose speeds are functions of the size of the problem, $\mathbf{s_i=f_i(x)}$, and (3) There is a limit $\mathbf{l_i}$ on the largest problem size that can be solved on each processor,
- Partition the problem into $\mathbf{p}$ disjoint sub-problems $\mathbf{x_i}$ (i=0,…,$\mathbf{p}$-1) such that (1) The size of the sub-problem $\mathbf{x_i}$ is proportional to the speed of the processor $\mathbf{i}$, and (2) The size of the sub-problem $\mathbf{x_i}$ is less than or equal to the limit $\mathbf{l_i}$ on the largest problem size that can be solved on each processor ($\mathbf{x_i} \le \mathbf{l_i}$).

In the presented research we do not take account of communication cost. Although we well understand the importance of its incorporation in our performance model, this is just out of scope of this research. We also understand the importance of the problems of efficient building and maintaining of our model. These two problems are subjects of our current research.

Based on a natural classification of mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) solving problems on networks of heterogeneous computers, we suggest an API for partitioning these mathematical objects. These interfaces allow the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition their mathematical objects. These partitioning interfaces are designed to be used along with various programming tools for parallel and distributed computing on heterogeneous networks.

Currently we have implemented only the set and dense matrix partitioning API of HDPI. In the next chapter, we present HMPI application programming that demonstrates how to write real-life HMPI applications using the extensions to MPI and the HDPI API and how to execute these applications.

CHAPTER 5

HMPI Application Programming

5.1 Example of irregular HMPI application

To explain how an application programmer can use HMPI to write a real-life irregular application, consider the EM3D application simulating the interaction of electric and magnetic fields on a three-dimensional object [YWC+95, CDG+93]. The system consists of a few large subbodies resulting from a decomposition of the three-dimensional object. The subbodies contain varying number of E nodes where electric field values are calculated and H nodes where magnetic fields are calculated. The changes in the electric field of an E node are calculated as a linear function of the magnetic field values of its neighboring H nodes and vice versa. Thus, the dependencies between E and H nodes form a bipartite graph. In a bipartite graph, the nodes are decomposed into two disjoint sets such that no two nodes within the same set are adjacent. Here the two disjoint sets are the set of E nodes and the set of H nodes. The subbodies are so decomposed from the three-dimensional object that the nodes in each subbody have few dependencies on the nodes residing in other subbodies thereby reducing the communications between a pair of subbodies. A sample decomposition of a three dimensional object into three subbodies is shown in Figure 5.1(a). A simple example of bipartite graph is shown in Figure 5.1(b).

The parallel algorithm of this application consists of a few parallel processes, each of which updates data characterizing a single sub-body. The heterogeneous algorithm can be summarized as follows:

- At each step of the algorithm,

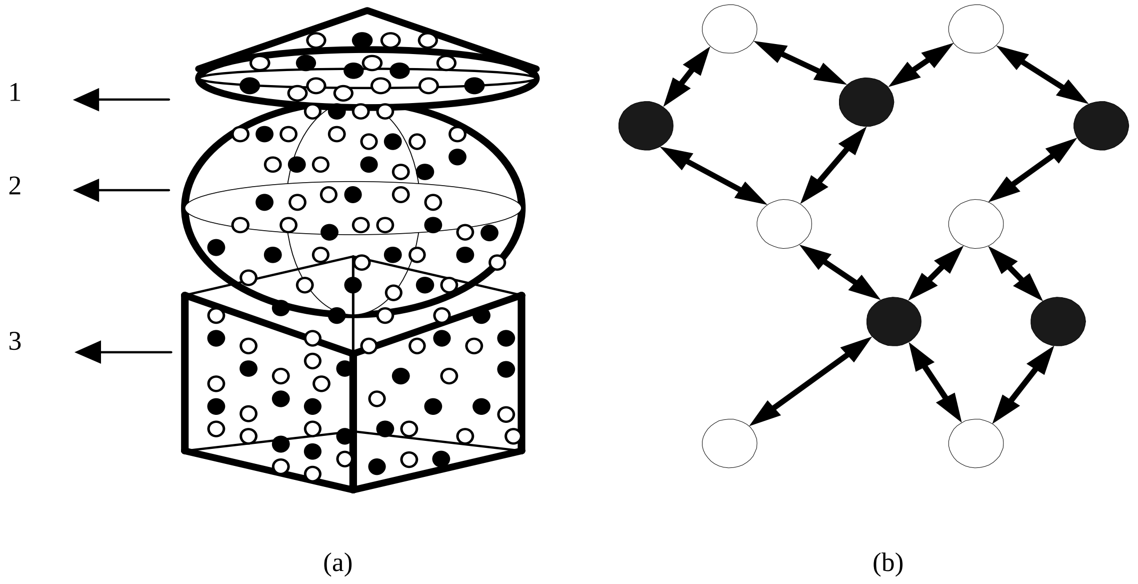


Figure 5.1: (a) A sample three dimensional object consists of three subbodies. In each subbody, the electric field value is represented as a white dot, an E node, and the magnetic field value represented by a black dot, an H node. (b) A bipartite graph showing the dependencies between E and H nodes.

- For each of the E nodes in its sub-body, if any of the neighboring H nodes reside remotely, each process receives the values of these nodes from the process owning them;
- Each process in parallel computes the new value of the electric field of each of the E nodes in its sub-body;
- For each of the H nodes in its sub-body, if any of the neighboring E nodes reside remotely, each process receives the values of these nodes from the process owning them;
- Each process in parallel computes the new value of the magnetic field of each of the H nodes in its sub-body.

```
int main(int argc, char **argv) {
   MPI_Comm em3dcomm;
   int i, me, is_executing_algo = MPI_UNDEFINED, E = 0, H = 1;
   int p, niter;                /* Inputs to the program */
   struct EM3D_body_t* bodies;  /* Inputs to the program */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &me);
   if (me >= 0 && me < p) is_executing_algo = 1;
   MPI_Comm_split(MPI_COMM_WORLD, is_executing_algo, 1, &em3dcomm);
   if (is_executing_algo) {
     Initialize_system(p, bodies);
     MPI_Comm_rank(&em3dcomm, &me);
     for (i = 0; i < niter; i++) {
       Gather_remote_H_boundary_values(me, H, p, bodies, &em3dcomm);
       Compute_E_values(me, E, p, bodies);
       Gather_remote_E_boundary_values(me, E, p, bodies, &em3dcomm);
       Compute_H_values(me, H, p, bodies);
     }
     MPI_Comm_free(&em3dcomm);
   }
   MPI_Finalize();
}
```

Figure 5.2: The most relevant fragments of code of the MPI program implementing the EM3D algorithm.

```
algorithm Em3d(int p, int k, int d[p], int dep[p][p]) {
  coord I=p;
  node {I>=0: bench*(d[I]/k);};
  link (L=p) {
    I>=0 && I!=L && (dep[I][L] > 0) :
      length*(dep[I][L]*sizeof(double)) [L]->[I];
  };
  parent[0];
  scheme {
    int current, owner, remote;
    par (owner = 0; owner < p; owner++)
        par (remote = 0; remote < p; remote++)
            if ((owner != remote) && (dep[owner][remote] > 0))
               100%%[remote]->[owner];
    par (current = 0; current < p; current++) 100%%[current];
  };
 }
```

Figure 5.3: Specification of the performance model of the EM3D algorithm in the HMPI's performance definition language.

The most interesting fragments of the MPI version of this parallel application are shown in Figure 5.2.

As shown in the MPI program above, the participating parallel processes in the group associated with the MPI communicator **`em3dcomm`** are explicitly chosen from an ordered set of processes specified by the group associated with the MPI communicator **`MPI_COMM_WORLD`**. If the MPI application runs on a homogeneous distributed-memory computer system, this group will execute the parallel algorithm with the same execution time as any other MPI group of processes, just because all processors run at the same speed, and all communication links transfer data at the same speed. However, if the MPI program runs on a HNOC, this group will execute the parallel algorithm sometimes slower and sometimes faster than other groups of processes. This is because different processors of the HNOC will execute the same computations at different speeds, and different pair of processors will communicate at different speeds. MPI does not facilitate creation of a group of processes where the processes are optimally selected taking into account the speeds of the processes, and the latencies and the bandwidths of the communication links between them. It is only a pure chance if the MPI group of processes executes the parallel algorithm faster than any other MPI group of processes on the HNOC.

If there is more than one process per processor, the first **`p`** processes are used to execute the MPI application. However, the HMPI application will select an optimal set of processes consisting of **`p`** processes dropping the rest of the processes from the computation when their participation can degrade performance. The MPI communicator **`em3dcomm`** represents this optimal set of processes in the HMPI application whereas it consists of first **`p`** processes from the pre-defined MPI communication universe **`MPI_COMM_WORLD`** in the corresponding MPI application.

The HMPI version of this parallel application involves first describing the performance model of the parallel algorithm. The definition of **`Em3d`** shown in Figure 5.3 describes the performance model of the heterogeneous algorithm of this parallel application.

The model describing the algorithm has 4 parameters:

- Parameter **`p`** specifies the number of abstract processors executing the algorithm;
- Parameter **`k`** specifies the number of nodes in a single subbody, whose data is computed in the benchmark code that is truly representative of the underlying application;
- It is supposed that **`i`**-th element of the vector parameter **`d`** gives the number of nodes in the subbody computed by the **`i`**-th abstract processor participating in the execution of the algorithm;
- Parameter **`dep`** specifies the number of nodal values communicated between different pairs of subbodies: **`dep[I][J]`** gives the number of nodal values in the subbody **`J`** that subbody **`I`** needs to compute its nodal values.

The **`coord`** declaration introduces one coordinate variable **`I`** ranging from **`0`** to **`p-1`**.

The **`node`** declaration associates the abstract processors with this coordinate system to form a linear processor arrangement. It also describes the absolute volume of computation to be performed by each of the processors. As a unit of measurement, the volume of computation performed by some benchmark code is used. In this particular case, it is assumed that the benchmark code computes the nodal values of **`k`** nodes in a single subbody. At each step of the algorithm, abstract processor P_I updates **`d[I]`** nodes. As computations during the updating of one single subbody mainly falls into the calculation of nodal values, the volume of computations performed by the abstract processor P_I will be approximately **`d[I]/k`** times larger than the volume of computations performed by the benchmark code.

The **`link`** declaration specifies the volumes of data to be transferred between the abstract processors at each step of the algorithm. Abstract processor P_I owning subbody **`I`** receives **`dep[I][L]`** remote boundary values from the subbody **`L`** owned by processor P_L. Thus, the total volume of data to be transferred from P_L to P_I will be equal to **`dep[I][L]*sizeof(double)`**.

```
int main(int argc, char **argv) {
   MPI_Comm em3dcomm;
   int i, me, k, E = 0, H = 1;
   HMPI_Group gid;
   void* model_params;
   int p, niter;                /* Inputs to the program */
   struct EM3D_body_t* bodies;  /* Inputs to the program */
   HMPI_Init(argc, argv);
   if (HMPI_Is_member(HMPI_PROC_WORLD_GROUP)) {
      int output_p;
      Body recon_body;
      // Construct recon parameters that are
      // representative of the application
      ...
      HMPI_Recon(&Serial_em3d, &recon_body, 1, &output_p);
   }
   if (HMPI_Is_host()) {
      HMPI_Pack_model_parameters(p, k, d, dep, model_params);
      HMPI_Group_create(&gid, &HMPI_Model_Em3d, model_params);
   }
   if (HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_Em3d, NULL);
   if (HMPI_Is_member(&gid)) {
     em3dcomm = *(MPI_Comm*)HMPI_Get_comm(&gid);
     Initialize_system(p, bodies);
     MPI_Comm_rank(&em3dcomm, &me);
     for (i = 0; i < niter; i++) {
       Gather_remote_H_boundary_values(me, H, p, bodies, &em3dcomm);
       Compute_E_values(me, E, p, bodies);
       Gather_remote_E_boundary_values(me, E, p, bodies, &em3dcomm);
       Compute_H_values(me, H, p, bodies);
     }
   }
   if (HMPI_Is_member(&gid)) HMPI_Group_free(&gid);
   HMPI_Finalize(0);
}
```

Figure 5.4: The most relevant code fragments of the HMPI program implementing the algorithm of EM3D.

The **`scheme`** declaration describes how the abstract processors interact during the execution of an iteration of the algorithm:

- Each processor P_{owner} first receives the remote values required for the calculation of the nodal values in its subbody. During this communication operation, 100% of data that should be sent from each processor P_{remote} to processor P_{owner} at this step will be sent. The second nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies it. The **`par`** algorithmic patterns are used to specify that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.
- Each processor then computes the new values for each of the nodes in its subbody. The processor will perform 100% of computations it should perform during this iteration. The **`par`** algorithmic patterns are used here to specify that all abstract processors perform their computations in parallel.

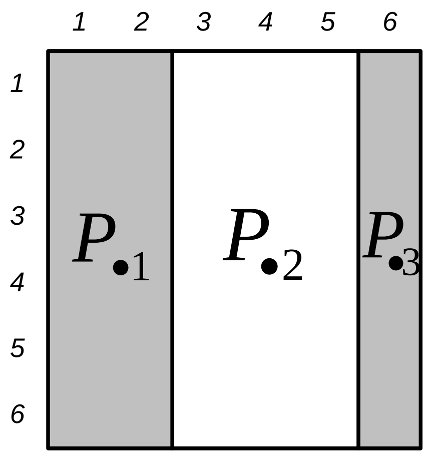
Note that the above performance model describes only one iteration of the algorithm. This approximation is accurate enough because at any iteration each processor performs the same volume of computations, and the same volume of data is transferred between each pair of processors.

The most interesting code fragments of the HMPI parallel application are shown in Figure 5.4. The HMPI runtime system is initialized using operation **`HMPI_Init`**. Then, operation **`HMPI_Recon`** updates the estimation of performances of processors using the serial EM3D program computing nodal values for a single subbody. The computations performed by each processor mainly fall into the execution of calls to function **`Serial_em3d`**.

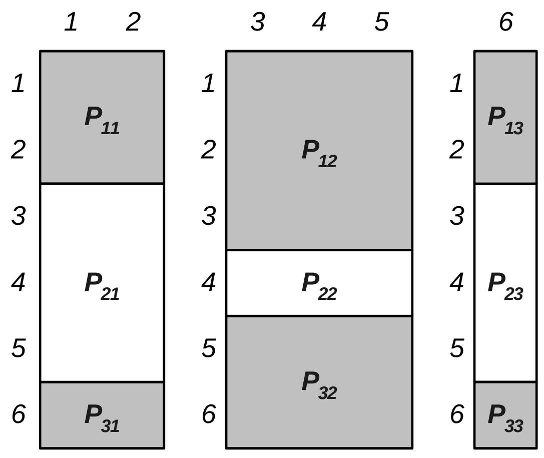
This is followed by the creation of a group of processes using operation **`HMPI_Group_create`**. The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **`HMPI_Group_free`**, and by finalizing the HMPI runtime system using operation **`HMPI_Finalize`**.

On HNOCs, the running time of the HMPI program shown above will normally be less than the running time of the corresponding MPI program. This is because an HMPI group of processes is created to execute the parallel algorithm faster than any other group of processes including the groups of processes created using MPI means. The processes participating in the HMPI group are chosen to minimize the execution time of the algorithm taking into account all its main features, which have an impact on the application execution performance. The application programmer describes all the main features of the parallel algorithm using the performance model **Em3d**, which are:

- The total number of participating processes **`p`**;
- The total volume of computations to be performed by each of the processes as specified in **node** declaration. The volume of computations is mainly the computation of field values of nodes in a sub-body thus depending on the number of nodes within a sub-body;
- The total volume of data to be transferred between each pair of processes as specified by the **link** declaration. The volume of data transferred equals the number of bytes of remote boundary values communicated between the sub-bodies;
- How exactly the processes interact during the execution of the algorithm as specified by the **scheme** declaration. Informally this looks like the description of the algorithm describing the interaction between the processes during the execution of the algorithm.



(a) Partition between processor columns.



(b) Partition inside each processor column.

Figure 5.5: Example of two-step distribution of a 6×6 generalized block over a 3×3 processor grid. The relative speed of processors is given by matrix $s=\begin{pmatrix}0.11 & 0.25 & 0.05\\ 0.17 & 0.09 & 0.08\\ 0.05 & 0.17 & 0.03\end{pmatrix}$. (a) At the first step, the 6×6 square is distributed in a one-dimensional block fashion over processor columns of the 3×3 processor grid in proportion $0.33:0.51:0.16 \approx 2:3:1$. (b) At the second step, each vertical rectangle is distributed independently in a one-dimensional block fashion over the processors of its column. The first rectangle is distributed in proportion $0.11:0.17:0.05 \approx 2:3:1$. The second one is distributed in proportion $0.25:0.09:0.17 \approx 3:1:2$. The third is distributed in proportion $0.05:0.08:0.03 \approx 2:3:1$.

During the creation of the group of processes, the HMPI runtime system uses the information from the performance model to solve the problem of selection of the optimal set of processes running on different computers of a heterogeneous network.

It can also be seen from the MPI and HMPI programs described in this section that there is essentially no change in code of the parallel algorithm executed by the members of the group of

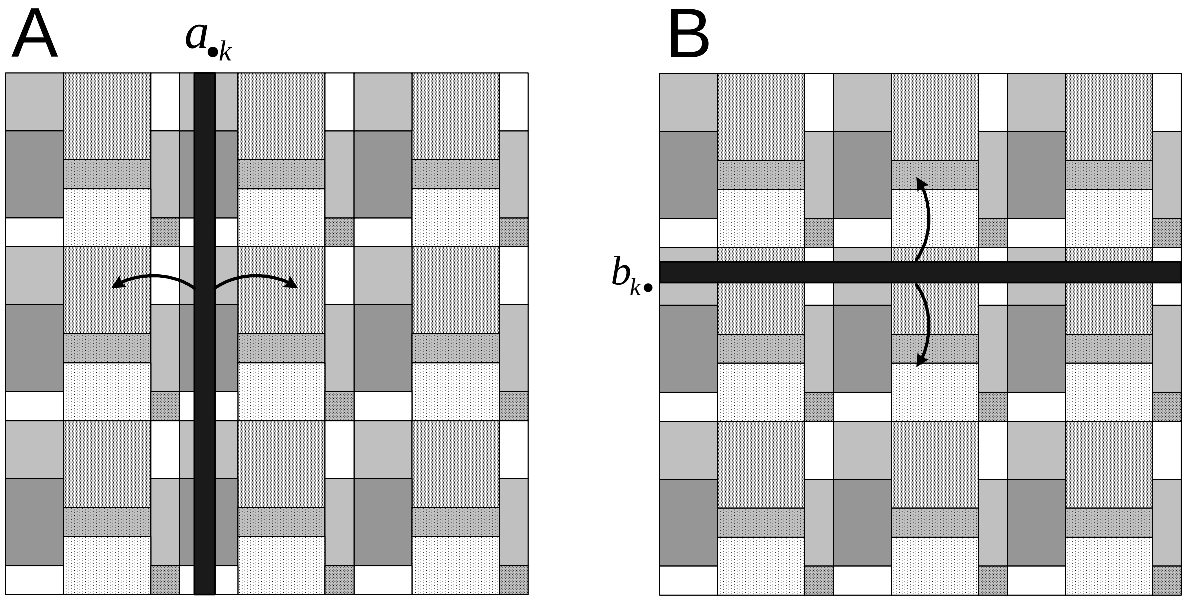


Figure 5.6: One step of the algorithm of parallel matrix-matrix multiplication based on heterogeneous two-dimensional block distribution of matrices A, B, and C. First, each $r \times r$ block of the pivot column $a_{\bullet k}$ of matrix A (shown shaded dark grey) is broadcast horizontally, and each $r \times r$ block of the pivot row $b_{k\bullet}$ of matrix B (shown shaded dark grey) is broadcast vertically.

processes participating in the parallel program. The main difference lies only in the creation of a group of processes.

5.2 Examples of regular HMPI application

An irregular problem is characterized by some inherent coarse-grained or large-grained structure implying quite deterministic decomposition of the whole program into a set of processes running in parallel and interacting via message passing. As a rule, there are essential differences in volumes of computations and communications to perform by different processes. The EM3D problem is an example of an irregular problem.

Unlike an irregular problem, for a regular problem the decomposition of the whole program into a large set of small equivalent programs, running in parallel and interacting via message passing, is the most natural one. Multiplication of dense matrices is an example of a regular problem. The main idea of efficiently solving a regular problem is to reduce it to such an irregular problem, the structure of which is determined by the irregularity of underlying hardware rather than the irregularity of the problem itself. So, the whole program is decomposed into a set of programs, each made from a number of the small equivalent programs stuck together and running on a separate processor of the underlying hardware.

Matrix Multiplication

Consider the problem of parallel matrix multiplication (MM) on HNOCs. The algorithm for the matrix operation C=A×B on a HNOC is obtained by modification of the ScaLAPACK [CDD+96] 2D block-cyclic MM algorithm. The modification is that the heterogeneous 2D block-cyclic data distribution of [KL01] is used instead of the standard homogeneous data distribution. Thus, the heterogeneous algorithm of multiplication of two dense square $(\mathbf{n}\times\mathbf{r})\times(\mathbf{n}\times\mathbf{r})$ matrices A and B on an $\mathbf{m}\times\mathbf{m}$ grid of heterogeneous processors can be summarised as follows:

- Each element in A, B, and C is a square $\mathbf{r}\times\mathbf{r}$ block and the unit of computation is the updating of one block, i.e., a matrix multiplication of size $\mathbf{r}$. Each matrix is partitioned into generalized blocks of the same size $(\mathbf{l}\times\mathbf{r})\times(\mathbf{l}\times\mathbf{r})$, where $\mathbf{m}\leq\mathbf{l}\leq\mathbf{n}$. The generalized blocks are identically partitioned into $\mathbf{p}^2$ rectangles, each being assigned to a different processor. The area of each rectangle is proportional to the speed of the processor that stores the rectangle. The partitioning of a generalized block is performed as follows:

  - Each element in the generalized block is a square **r**×**r** block of matrix elements. The generalized block is an **l**×**l** square of **r**×**r** blocks.
  - First, the **l**×**l** square is partitioned into **m** vertical slices, so that the area of the *j*-th slice is proportional to $\sum_{i=1}^{m} s_{ij}$ (see Figure 5.5(a)). It is supposed that blocks of the *j*-th slice will be assigned to processors of the *j*-th column in the **m**×**m** processor grid. Thus, at this step, we balance the load between processor columns in the **m**×**m** processor grid, so that each processor column will store a vertical slice whose area is proportional to the total speed of its processors.
  - Then, each vertical slice is partitioned independently into **m** horizontal slices, so that the area of the *i*-th horizontal slice in the *j*-th vertical slice is proportional to s_{ij} (see Figure 5.5(b)). It is supposed that blocks of the *i*-th horizontal slice in the *j*-th vertical slice will be assigned to processor P_{ij}. Thus, at this step, we balance the load of processors within each processor column independently.
- At each step **k**,
  - Each **r**×**r** block a_{ik} of the pivot column of matrix A is sent horizontally from the processor, which stores this block, to **m**-1 processors (see Figure 5.6);
  - Each **r**×**r** block b_{kj} of the pivot row of matrix B is sent vertically from the processor, which stores this block, to **m**-1 processors (see Figure 5.6);
- Each processor updates its rectangle in the C matrix with one block from the pivot row and one block from the pivot column.

```
typedef struct {int I; int J;} Processor;
algorithm ParallelAxB(int m, int r, int n, int l, int w[m],
                    int h[m][m][m][m])
{
  coord I=m, J=m;
  node {I>=0 && J>=0: bench*(w[J]*(h[I][J][I][J])*(n/l)*(n/l)*n);};
  link (K=m, L=m)
  {
    I>=0 && J>=0 && I!=K :
      length*(w[J]*(h[I][J][I][J])*(n/l)*(n/l)*(r*r)*sizeof(double))
             [I, J] -> [K, J];
    I>=0 && J>=0 && J!=L && ((h[I][J][K][L])>0) :
      length*(w[J]*(h[I][J][K][L])*(n/l)*(n/l)*(r*r)*sizeof(double))
             [I, J] -> [K, L];
  };
  parent[0,0];
  scheme
  {
    int k, *w, *h, *trow, *tcol;
    Get_trow_tcol(m, w, h, trow, tcol);
    Processor Root, Receiver, Current;
    for(k = 0; k < n; k++)
    {
      int Acolumn = k%l, Arow;
      int Brow = k%l, Bcolumn;
      par(Arow = 0; Arow <l; )
      {
        Get_matrix_processor(Arow, Acolumn, p, q, w, h, trow, tcol, &Root);
        par(Receiver.I = 0; Receiver.I < m; Receiver.I++)
          par(Receiver.J = 0; Receiver.J < m; Receiver.J++)
            if((Root.I != Receiver.I || Root.J != Receiver.J) &&
               Root.J != Receiver.J)
              if((h[Root.I][Root.J][Receiver.I][Receiver.J]) > 0)
                (100.00/(w[Root.J]*(n/l)))%%
                      [Root.I, Root.J] -> [Receiver.I, Receiver.J];
        Arow += h[Root.I][Root.J][Root.I][ Root.J];
      }
      par(Bcolumn = 0; Bcolumn < l; )
      {
        Get_matrix_processor(Brow, Bcolumn, p, q, w, h, trow, tcol, &Root);
        par(Receiver.I = 0; Receiver.I < m; Receiver.I++)
          if(Root.I != Receiver.I)
            (100.00/((h[Root.I][Root.J][Root.I][Root.J])*(n/l))) %%
                  [Root.I, Root.J] -> [Receiver.I, Root.J];
        Bcolumn += w[Root.J];
      }
      par(Current.I = 0; Current.I < m; Current.I++)
        par(Current.J = 0; Current.J < m; Current.J++)
          (100.00/n) %% [Current.I, Current.J];
    }
  };
};
```

Figure 5.7: Specification of the performance model of the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices in the HMPI’s performance definition language.

The definition of **`ParallelAxB`** given in Figure 5.7 describes the performance model of this heterogeneous algorithm.

The performance model **`ParallelAxB`** describing the algorithm has 6 parameters. Parameter **`m`** specifies the number of abstract processors along the rows and along the columns of the processor grid executing the algorithm. Parameter **`r`** specifies the size of a square block of matrix elements, the updating of which is the unit of computation of the algorithm. Parameter **`n`** is the size of square matrices *A*, *B*, and *C* measured in **`r`**×**`r`** blocks. Parameter **`l`** is the size of a generalised block also measured in **`r`**×**`r`** block.

Vector parameter **`w`** specifies the widths of the rectangles of a generalised block assigned to different abstract processors of the **`m`**×**`m`** grid. The width of the rectangle assigned to processor P_{IJ} is given by element **`w[J]`** of the parameter (see Figure 5.5). All widths are measured in **`r`**×**`r`** blocks.

Parameter **`h`** specifies the heights of rectangle areas of a generalised block of matrix *A*, which are horizontally communicated between different pairs of abstract processors. Let R_{IJ} and R_{KL} be the rectangles of a generalised block of matrix *A* assigned to processors P_{IJ} and P_{KL} respectively. Then, **`h[I][J][K][L]`** gives the height of the rectangle area of R_{IJ}, which is required by processor P_{KL} to perform its computations. All heights are measured in **`r`**×**`r`** blocks.

Figure 4.18 illustrates possible combinations of rectangles R_{IJ} and R_{KL} in a generalised block. Let us call an **`r`**×**`r`** block of R_{IJ} a *horizontal neighbour* of R_{KL} if the row of **`r`**×**`r`** blocks that contains this **`r`**×**`r`** block will also contain an **`r`**×**`r`** block of R_{KL}. Then, the rectangle area of R_{IJ}, which is required by processor P_{KL} to perform its computations, comprises of all horizontal neighbours of R_{KL}.

Figure 4.18(a) shows the situation when rectangles R_{IJ} and R_{KL} have no horizontal neighbours. Correspondingly, **`h[I][J][K][L]`** will be zero.

Figure 4.18(b) shows the situation when all **`r`**×**`r`** blocks of R_{IJ} are horizontal neighbours of R_{KL}. In that case, both **`h[I][J][K][L]`** will be equal to the height of R_{IJ}.

Figures 4.18(c) and 4.18(d) show the situation when only some of the **`r`**×**`r`** blocks of R_{IJ} are horizontal neighbours of R_{KL}. In this case, **`h[I][J][K][L]`** will be equal to the height of the rectangle subarea of R_{IJ} comprising the horizontal neighbours of R_{KL}.

Note that **`h[I][J][I][J]`** specifies the height of R_{IJ}, and **`h[I][J][K][L]`** will always be equal to **`h[K][L][I][J]`**.

The **`coord`** declaration introduces 2 coordinate variables, **`I`** and **`J`**, both ranging from **`0`** to **`m-1`**.

The **`node`** declaration associates the abstract processors with this coordinate system to form an **`m`**×**`m`** grid. It also describes the absolute volume of computation to be performed by each of the processors. As a unit of measure, the volume of computation performed by the code multiplying two **`r`**×**`r`** matrices is used. At each step of the algorithm, abstract processor P_{IJ} updates $(w_{IJ} \times h_{IJ}) \times n_g$ **`r`**×**`r`** blocks, where w_{IJ}, h_{IJ} are the width and height of the rectangle of a generalised block assigned to processor P_{IJ}, and n_g is the total number of generalised blocks. As computations during the updating of one **`r`**×**`r`** block mainly fall into the multiplication of two **`r`**×**`r`** blocks, the volume of computations performed by the processor P_{IJ} at each step of the algorithm will be approximately $(w_{IJ} \times h_{IJ}) \times n_g$ times larger than the volume of computations performed to multiply two **`r`**×**`r`** matrices. w_{IJ} is given by **`w[J]`**, h_{IJ} is given by **`h[I][J][I][J]`**, n_g is given by **`(n/l)*(n/l)`**, and the total number of steps of the algorithm is given by **`n`**. Therefore the total volume of computation performed by abstract processor P_{IJ} will be **`w[J]*h[I][J][I][J]*(n/l)*(n/l)*n`** times bigger than the volume of computation performed by the code multiplying two **`r`**×**`r`** matrices.

The **`link`** declaration specifies the volumes of data to be transferred between the abstract processors during the execution of the algorithm. The first statement in this declaration describes communications related to matrix *A*. Obviously, abstract processors from the same column of the processor grid do not send each other elements of matrix *A*. Abstract processor P_{IJ} will send elements of matrix *A* to processor P_{KL} only if its rectangle R_{IJ} in a generalised block has horizontal neighbours of the rectangle R_{KL} assigned to processor P_{KL}. In that case, processor P_{IJ} will send all such neighbours to processor P_{KL}. Thus, in total processor P_{IJ} will send $N_{IJKL} \times n_g$ **`r`**×**`r`** blocks of matrix *A* to processor P_{KL}, where N_{IJKL} is the number of horizontal neighbours of rectangle R_{KL} in rectangle R_{IJ}, and n_g is the total number generalised blocks. N_{IJKL} is given by **`w[J]* h[I][J][K][L]`**, n_g is given by **`(n/l)*(n/l)`**, and the volume of data in one **`r`**×**`r`** block is given by **`(r*r)*sizeof(double)`**. Therefore the total volume of data transferred from processor P_{IJ} to processor P_{KL} will be given by **`w[J]*h[I][J][K][L]*(n/l)*(n/l)*(r*r)*sizeof(double)`**.

The second statement in the **`link`** declaration describes communications related to matrix *B*. Obviously, only abstract processors from the same column of the processor grid send each other elements of matrix *B*. In particular, processor P_{IJ} will send all its **`r`**×**`r`** blocks of matrix *B* to all other processors from column *J* of the processor grid. The total number of **`r`**×**`r`** blocks of matrix *B* assigned to processor P_{IJ} is given by **`w[J]*h[I][J][I][J]*(n/l)*(n/l)`**.

The **`scheme`** declaration describes **`n`** successive steps of the algorithm. At each step **`k`**,

- A row of **`r`**×**`r`** blocks of matrix *B* is communicated vertically. For each pair of abstract processors P_{IJ} and P_{KJ} involved in this communication, P_{IJ} sends a part of this row to P_{KJ}. The number of **`r`**×**`r`** blocks transferred from P_{IJ} to P_{KJ} will be $w_{IJ} \times \sqrt{n_g}$, where $\sqrt{n_g}$ is the number of generalised blocks along the row of **`r`**×**`r`** blocks. The total number of **`r`**×**`r`** blocks of matrix *B*, which processor P_{IJ} sends to processor P_{KJ}, is $(w_{IJ} \times h_{IJ}) \times n_g$. Therefore, $\frac{w_{IJ} \times \sqrt{n_g}}{(w_{IJ} \times h_{IJ}) \times n_g} \times 100 = \frac{1}{h_{IJ} \times \sqrt{n_g}} \times 100$ percent of all data that should be in total sent from processor P_{IJ} to processor P_{KJ} will be sent at the step. The first nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies this fact. The **`par`** algorithmic patterns are used to specify that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.
- A column of **`r`**×**`r`** blocks of matrix *A* is communicated horizontally. If processors P_{IJ} and P_{KL} are involved in this communication so that P_{IJ} sends a part of this column to P_{KL}, then the number of **`r`**×**`r`** blocks transferred from P_{IJ} to P_{KL} will be $H_{IJKL} \times \sqrt{n_g}$, where H_{IJKL} is the height of the rectangle area in a generalised block, which is communicated from P_{IJ} to P_{KL}, and $\sqrt{n_g}$ is the number of generalised blocks along the column of **`r`**×**`r`** blocks. The total number of **`r`**×**`r`** blocks of matrix *A*, which processor P_{IJ} sends to processor P_{KL}, is $N_{IJKL} \times n_g$. Therefore, $\frac{H_{IJKL} \times \sqrt{n_g}}{N_{IJKL} \times n_g} \times 100 = \frac{H_{IJKL} \times \sqrt{n_g}}{(H_{IJKL} \times w_{IJ}) \times n_g} \times 100 = \frac{1}{w_{IJ} \times \sqrt{n_g}} \times 100$ percent of all data that should be in total sent from processor P_{IJ} to processor P_{KL} will be sent at the step.

The second nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration specifies this fact. Again, we use the **`par`** algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.

- Each abstract processor updates each of its **`r`**×**`r`** block of matrix *C* with one block from the pivot column and one block from the pivot row, so that each block c_{ij} $(i, j \in \{1, \ldots, n\})$ of matrix *C* will be updated to have the values $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$. The processor performs the same volume of computation at each step of the algorithm. Therefore, at each of **`n`** steps of the algorithm the processor will perform $\frac{100}{n}$ percent of the volume of computations it performs during the execution of the algorithm. The third nested **`par`** statement in the main **`for`** loop of the **`scheme`** declaration just specifies this fact. The **`par`** algorithmic patterns are used here to specify that all abstract processors perform their computations in parallel.

The function **`Get_matrix_processor`** used in the **`scheme`** declaration is a matrix partitioning API, which is part of the Heterogeneous Data Partitioning Interface (HDPI) discussed in Chapter 4. It is used to iterate over abstract processors that store the pivot row and the pivot column of **`r`**×**`r`** blocks. It returns in its last parameter the grid coordinates of the abstract processor storing the **`r`**×**`r`** block, whose coordinates in a generalised block of a matrix are specified by its first two parameters.

The performance model **`ParallelAxB`** shown in the Figure 5.7 is applicable to the heterogeneous algorithm with **`COLUMN_BASED`** data distribution. However this model can be

```
int m, l, r, n;
int main(int argc, char** argv) {
    int optimal_generalised_block_size;
    int *w, *h; //Matrix partitioning parameters
    typedef struct {double *a; double *b; double *c; int r;}
            Recon_params;
   HMPI_Group gid;
   void *model_params;
   double *a, *b, *c;

   HMPI_Init(argc, argv);
    if (HMPI_Is_member(HMPI_PROC_WORLD_GROUP)) {
      int output_p;
      Recon_params recon_params;
      Initialize(a, b, c, r, &recon_params);
      HMPI_Recon(&rMxM, &recon_params,  1, &output_p);
    }
    if (HMPI_Is_host()) {
      int bsize;
      double time, min_time=DBL_MAX;
      for (bsize = m; bsize < n; bsize++) {
          Partition_matrix_2d(
             p, q, 1, speeds, NULL, NULL, bsize, bsize, COLUMN_BASED,
             w, h, NULL, NULL, NULL, NULL);
          HMPI_Pack_model_parameters(p, q, n, bsize, r, w, h, model_params);
          time = HMPI_Timeof(&HMPI_Model_ParallelAxB, model_params);
          if (time < min_time) {
             optimal_generalised_block_size = bsize;
             min_time = time;
          }
      }
   }
   …
   l = optimal_generalised_block_size;
   if (HMPI_Is_host()) {
      HMPI_Pack_model_parameters(p, q, n, l, r, w, h, model_params);
      HMPI_Group_create(&gid, &HMPI_Model_ParallelAxB, model_params);
   }
   if (HMPI_Is_free())
      HMPI_Group_create(&gid, &HMPI_Model_ParallelAxB, NULL);
   if (HMPI_Is_member(&gid)) {
      …
      MPI_Comm* grid_comm = (MPI_Comm*)HMPI_Get_comm(&gid);
      …
      // computations and communications are performed here
      // using standard MPI routines.
      //
      …
   }
   if (HMPI_Is_member(&gid)) HMPI_Group_free(&gid);
   HMPI_Finalize(0);
}
```

Figure 5.8: The core of the HMPI program implementing the algorithm of parallel matrix multiplication based on heterogeneous two-dimensional block-cyclic distribution of matrices.

```
typedef struct {int I; int J;} Processor;
algorithm ParallelCholesky(int m, int r, int n, int l, int w[m],
                           int h[m][m][m][m]) {
  coord I=m, J=m;
  node {I>=0 && J>=0: bench*(Get_my_elements(I, J, m, m, w, h,
                                          NULL, NULL, COLUMN_BASED, ‘L’)*n);};
  link (K=m, L=m) {
    I>=0 && J>=0 && I!=K && J==L:
      length*(Get_diagonal(I, J, m, m, w, h, NULL, NULL, COLUMN_BASED)
            *(n/l-1)*r*r*sizeof(double))  [I, J] -> [K, L];
    I>=0 && J>=0 && I<K && J==L:
      length*(Get_diagonal(I, J, m, m, w, h,
              NULL, NULL, COLUMN_BASED)*r*r*sizeof(double)) [I, J] -> [K, L];
    I>=0 && J>=0 && J!=L:
      length*(Get_my_elements(I, J, m, m, w, h,
              NULL, NULL, COLUMN_BASED, ‘L’)*r*r*sizeof(double)) [I, J] -> [K, L];
  };
  parent[0,0];
  scheme {
    int i, k, *w, *h, *trow, *tcol; Processor Root, Roots, Receiver, Current;
    Get_trow_tcol(m, w, h, trow, tcol);
    for(k = 0; k < n; k++) {
       Get_matrix_processor(k%l, k%l, m, m, w, h, trow, tcol, &Root);
       (100.00/(Get_my_elements(Root.I, Root.J, m, m, w, h,
                NULL, NULL, COLUMN_BASED, ‘L’)*n)) %% [Root.I, Root.J];
       if ((k+1) == n) break;
       par(Receiver.I = 0; Receiver.I < m; Receiver.I++)
         par(Receiver.J = 0; Receiver.J < m; Receiver.J++)
           if (Root.J == Receiver.J)
             if (((k < (n-l)) && (Root.I != Receiver.I))
                 || ((k >= (n-l)) && (Root.I < Receiver.I)))
                (100.00/(Get_diagonal(Root.I, Root.J, m, m, w, h, NULL, NULL,COLUMN_BASED)
                *(n/l))) %% [(Root.I), (Root.J)] -> [(Receiver.I), (Receiver.J)];
       par (Current.I = 0; Current.I < m; Current.I++)
          par (Current.J = 0; Current.J < m; Current.J++)
             if (Current.J == Root.J) {
               int e = Get_my_elements(Current.I, Current.J, m, m, w, h, NULL, NULL,
                       COLUMN_BASED, ‘L’)*n*r;
               if ((k < (n-l)) || (Current.I > Root.I))
                  (100.00*(1 + log(r))/e) %% [Current.I, Current.J];
             }
       par (j = k+1; j < n; j++) {
         Get_matrix_processor(j%l, k%l, m, m, w, h, trow, tcol, &Roots);
         (100.00*(r*r + r*r)/(Get_my_elements(Roots.I, Roots.J, m, m, w, h, NULL, NULL,
                              COLUMN_BASED, ‘L’)*n*r*r*r)) %% [(Roots.I), (Roots.J)];
      }
       par (j = k+1; j < n; j++) {
          Get_matrix_processor(k%l, k%l, m, m, w, h, trow, tcol, &Roots);
          par (Receiver.I = 0; Receiver.I < m; Receiver.I++)
            par (Receiver.J = 0; Receiver.J < m; Receiver.J++)
               if (Roots.J != Receiver.J)
                    (100.00/(Get_my_elements(Roots.I, Roots.J, m, m, w, h, NULL, NULL,
                    COLUMN_BASED, ‘L’)) %% [Roots.I, Roots.J] -> [Receiver.I, Receiver.J];
      }
       par (Current.I = 0; Current.I < m; Current.I++)
         par (Current.J = 0; Current.J < m; Current.J++)
          ((100.00*Get_my_kk_elements(k, Current.I, Current.J, m, m, w, h, NULL, NULL,
             COLUMN_BASED, ‘L’))
          /
          (Get_my_elements(Current.I, Current.J, m, m, w, h,
                           NULL, NULL, COLUMN_BASED, ‘L’)*n)) %% [Current.I, Current.J];
   }
  };
};
```

Figure 5.9: Specification of the performance model of the algorithm of parallel Cholesky factorization based on heterogeneous two-dimensional block-cyclic distribution of matrices in the HMPI’s performance definition language.

made generic and applicable for any type of data distribution by adding an extra parameter to its parameter list (the type of distribution) and using data partitioning API of HDPI in the body of the performance model. This extra parameter is the type of data distribution such as **`COLUMN_BASED`** or **`ROW_BASED`** or **`CARTESIAN`** or **`RECURSIVE`**.

```
algorithm ParallelAxB(int m, int r, int n, int l,
         int type_of_distribution, int speeds[m*m])
```

The most interesting fragments of the rest code of the HMPI parallel application are shown in Figure 5.8. The HMPI runtime system is initialised using operation **`HMPI_Init`**. Then, operation **`HMPI_Recon`** updates the estimation of performances of processors using the serial multiplication of test matrices of size **`r`**×**`r`**. The computations performed by each processor mainly fall into the execution of calls to function **`rMxM`**.

The next block of code, executed by the host-processor, uses operation **`HMPI_Timeof`** to predict the total time of execution of the parallel algorithm. This operation is used to calculate the optimal generalized block size, one of the parameters of the heterogeneous parallel algorithm.

This is followed by the creation of a group of processes using operation **`HMPI_Group_create`**. The members of this group then perform the computations and communications of the heterogeneous parallel algorithm using standard MPI means. This is followed by freeing the group using operation **`HMPI_Group_free`** and the finalization of HMPI runtime system using operation **`HMPI_Finalize`**.

Cholesky Factorization

Consider the problem of parallel Cholesky factorization on HNOCs.

Cholesky Factorization factors a symmetric, positive definite matrix A into a product of a lower triangular matrix L and its tranpose; i.e., $A = L \cdot L^t$. One can partition the matrices A, L, and L^t and write the system as

$$\begin{bmatrix} A_{11} & A_{21}^t \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} L_{11}^t & L_{21}^t \\ 0 & L_{22}^t \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^t & L_{11}L_{21}^t \\ L_{21}L_{11}^t & L_{21}L_{21}^t + L_{22}L_{22}^t \end{bmatrix}.$$

If L_{11}, the lower triangle Cholesky factor of A_{11} is known, then the block equations can be arranged as

$$L_{21} \leftarrow A_{21}\left(L_{11}^t\right)^{-1},$$

$$A_{22} \leftarrow A_{22} - L_{21}L_{21}^t = L_{22}L_{22}^t.$$

The factorization can be done recursively applying the steps outlined above to the updated matrix A_{22}.

The algorithm of execution of the factorization on a HNOC is obtained by modification of the ScaLAPACK [CDD+96] 2D block-cyclic Cholesky algorithm. The modification is that the heterogeneous 2D block-cyclic data distribution of [KL01] is used instead of the standard homogeneous data distribution. The heterogeneous algorithm of factorisation of a (**n**×**r**)×(**n**×**r**) matrix A on an **m**×**m** grid of heterogeneous processors can be summarised as follows:

1. Each element in A is a square **r**×**r** block. Each matrix is partitioned into generalized blocks of the same size (**l**×**r**)×(**l**×**r**), where **m**≤**l**≤**n**. The generalized blocks are identically partitioned into $\mathbf{p^2}$ rectangles, each being assigned to a different processor. The partitioning of a generalized block is shown in Figure 5.5(a) and 5.5(b);
2. The largest A_{11} belonging to one process is selected and this process computes L_{11};
3. L_{11} is broadcast to other processes of the grid column and the grid column processes compute L_{21};

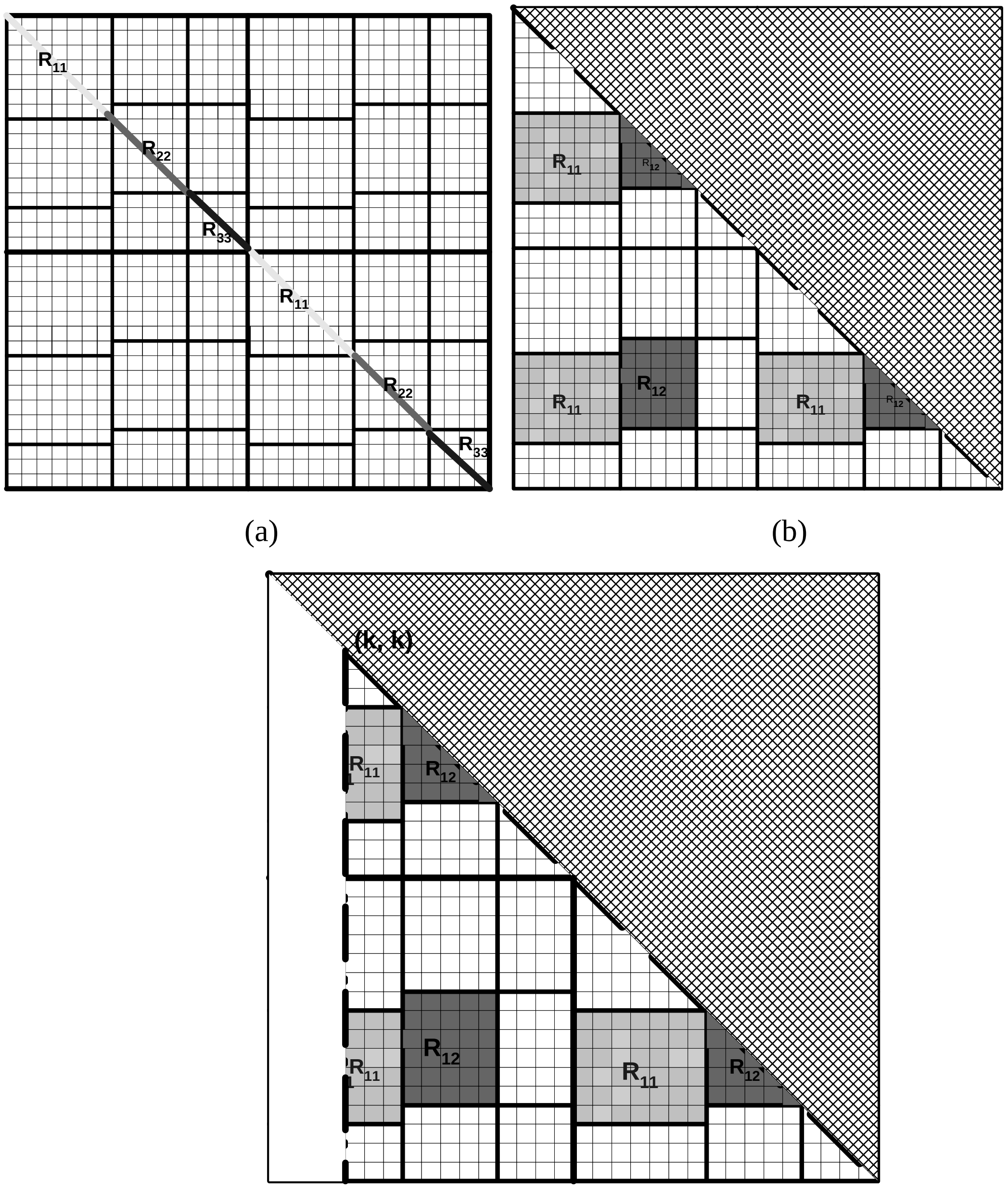
4. L_{21} is broadcast to processes of the other grid columns;
5. L_{21} is transposed using the broadcast values of the L_{21} on the grid columns;
6. All processes update A_{22}.

The definition of **`ParallelCholesky`** given in Figure 5.9 describes the performance model of this heterogeneous algorithm.

The performance model **`ParallelCholesky`** describing the algorithm has 6 parameters. These parameters hold the same meaning as those used in the performance model for matrix-matrix multiplication presented previously.

The **`coord`** declaration introduces 2 coordinate variables, **`I`** and **`J`**, both ranging from **`0`** to **`m`**-`1`.

The **`node`** declaration associates the abstract processors with this coordinate system to form a **`m`**×**`m`** grid. It also describes the absolute volume of computation to be performed by each of the processors. As a unit of measure, the volume of computation performed by the code factorizing an **`r`**×**`r`** matrix is used. Each abstract processor performs totally **`Get_my_elements(…)*n*r*r*r`** number of computations where the function **`Get_my_elements`** (part of the matrix partitioning API of HDPI) returns the number of elements owned by the processor in the lower triangular half of the factorized matrix including the diagonal elements. The purpose of this function is illustrated in Figure 5.10(b). Since the benchmark code factorizing an **`r`**×**`r`** matrix performs **`r`**×**`r`**×**`r`** computations, the total volume of computation performed by abstract processor P_{IJ} will be $\frac{Get\_my\_elements(...)\times n\times r\times r\times r}{r\times r\times r} = Get\_my\_elements(...)\times n$ times bigger than the volume of computation performed by the code factorizing an **`r`**×**`r`** matrix.



(c)

Figure 5.10. The matrix A consists of 4 generalized blocks of size 16×16. (a) The total number of r×r blocks along the diagonal owned by processors R_{11}, R_{22}, and R_{33} given by function **Get_diagonal**() are 7, 5, and 4 respectively. (b) The total number of r×r blocks given by the function **Get_my_elements**() belonging to processor owning the rectangle R_{11} are 42×3=126. The total number of r×r blocks belonging to processor owning the rectangle R_{12} are 30+2×15=60. (c) At step k, the total number of A_{22} r×r blocks given by the function **Get_my_kk_elements**() belonging to processor owning the rectangle R_{11} are 42+2×18=78. The total number of r×r blocks belonging to processor owning the rectangle R_{12} are 30+2×15=60.

The **`link`** declaration specifies the volumes of data to be transferred between the abstract processors during the execution of the algorithm. The first statement in this declaration describes communications related to broadcast of L_{11}. At each step **`k`** of the algorithm where $k < (n - l)$, the abstract processor P_{IJ} owning L_{11} broadcasts it to abstract processors P_{KJ} from the same column of the processor grid, i.e., abstract processors P_{KJ} where $\{I, K = 1 \cdots m, I \neq K\}$. The number of elements broadcast by each abstract processor P_{IJ} is equal to the number of elements owned by it on the diagonal of the matrix A. The number of **`r`**×**`r`** blocks owned by abstract processor P_{IJ} on the diagonal in a generalized block is given by function **`Get_diagonal`** (part of the matrix partitioning API of HDPI), which is illustrated in Figure 5.10(a). Since there are **`(n/l-1)`** generalized blocks (excluding the last generalized block), the total number of **`r`**×**`r`** blocks broadcast by abstract processor P_{IJ} is **`Get_diagonal(…)*(n/l-1)`**. The volume of data in one **`r`**×**`r`** block is given by **`(r*r)*sizeof(double)`**, so the total volume of data transferred from processor P_{IJ} to processor P_{KJ} will be given by **`Get_diagonal(…)*(n/l-1)*(r*r)*sizeof(double)`**.

The second statement in the **`link`** declaration describes communications of L_{11} where $k \geq (n - l)$, i.e., communications in the last generalized block. At each step **`k`** of the algorithm where $k \geq (n - l)$, the abstract processor P_{IJ} owning L_{11} broadcasts it to abstract processors P_{KJ} from the same column of the processor grid with coordinate K greater than the coordinate I of abstract processor P_{IJ}, i.e., abstract processors P_{KJ} where $\{I, K = 1 \cdots m, I < K\}$. The number of elements broadcast by each abstract processor P_{KJ} is equal to the number of elements owned by it on the diagonal of the factorized matrix. Since there is one generalized block, the total number of **`r`**×**`r`** blocks broadcast by abstract processor P_{IJ} is **`Get_diagonal(…)`** and since the volume of data in one **`r`**×**`r`** block is given by **`(r*r)*sizeof(double)`**, the total volume of data transferred from processor P_{IJ} to processor P_{KJ} will be given by **`Get_diagonal(…)*(r*r)*sizeof(double)`**.

The third statement in the **`link`** declaration describes broadcast of L_{21}. At each step **`k`** of the algorithm, the abstract processor P_{IJ} owning the elements of panel L_{21} broadcasts its elements to abstract processors P_{KL} from the other columns of the processor grid, i.e., abstract processors P_{KL} where $\{I, J, K, L = 1 \cdots m, J \neq L\}$. The number of elements broadcast by each abstract processor P_{KJ} is equal to the number of elements owned by it in the lower triangular half of the matrix A. The number of **`r`**×**`r`** blocks owned by abstract processor P_{IJ} in the lower triangular half of the factorized matrix is given by function **`Get_my_elements`**, which is illustrated in Figure 5.10(b). Since the volume of data in one **`r`**×**`r`** block is given by **`(r*r)*sizeof(double)`**, the total volume of data transferred from processor P_{IJ} to processor P_{KL} will be given by **`Get_my_elements(…)*(r*r)*sizeof(double)`**.

The **`scheme`** declaration describes **`n`** successive steps of the algorithm. At each step **`k`**,

- The lower triangle Cholesky factor L_{11} of A_{11} is computed by the processor **`Root`** owning the **`r`**×**`r`** block A_{11}. The total number of computations performed by each processor during the execution of the algorithm is equal to **`Get_my_elements(…)*n*r*r*r`**, so at each such step, the processor will perform $\frac{100 \times r \times r \times r}{Get\_my\_elements(...) \times n \times r \times r \times r} = \frac{100}{Get\_my\_elements(...) \times n}$ percent of the volume of computations it performs during the execution of the algorithm.
- The lower triangle Cholesky factor L_{11} of A_{11} is broadcast by processor P_{IJ} to processors P_{KJ} belonging to the same column of the processor grid where $\{I, K = 1 \cdots m, I \neq K\}$. At this step, processor P_{IJ} sends a volume of data equivalent to

(r*r)*sizeof(double) to P_{KJ}. Therefore $\frac{100 \times r \times r \times sizeof(double)}{Get\_diagonal(...) \times (\frac{n}{l}) \times r \times r \times sizeof(double)} = \frac{100}{Get\_diagonal(...) \times (\frac{n}{l})}$ percent of all data that should be sent from P_{IJ} to P_{KJ} is sent at this step. We use the **par** algorithmic patterns in this specification to stress that during the execution of this communication, data transfer between different pairs of processors is carried out in parallel.

- Each abstract processor then computes the inverse of the L_{11} that it received in the previous step followed by the transpose of the result. The transpose of an **r**×**r** block takes **r**×**r** number of computations and the inverse of an **r**×**r** block takes **r**×**r**×**log(r)** number of computations. Since the total number of computations performed by each processor during the execution of the algorithm is equal to **Get_my_elements(…)*n*r*r*r**, at each such step, the processor will perform $\frac{100 \times (r \times r + r \times r \times \log r)}{Get\_my\_elements(...) \times n \times r \times r \times r} = \frac{100 \times (1 + \log r)}{Get\_my\_elements(...) \times n \times r}$ percent of the volume of computations it performs during the execution of the algorithm.
- Each abstract processor multiplies its **r**×**r** blocks of A_{21} by $\left(L_{11}^t\right)^{-1}$ that it computed in the previous step to get L_{21}. This step consists of **n-(k+1)** sub-steps. At each such sub-step, the resulting **r**×**r** blocks of L_{21} from the multiplication of A_{21} by $\left(L_{11}^t\right)^{-1}$ are transposed. At each such sub-step, the total number of computations involved in the multiplication is **r**×**r** and the total number of computations involved in a transpose is **r**×**r**. Since the total number of computations performed by each processor is equal to **Get_my_elements(…)*n*r*r*r**, at each such sub-step, the processor will perform

$\frac{100\times(r\times r+r\times r)}{Get\_my\_elements(...)\times n\times r\times r\times r}=\frac{100\times 2}{Get\_my\_elements(...)\times n\times r}$ percent of the volume of computations it performs during the execution of the algorithm.

- Each abstract processor P_{IJ} then broadcasts its **`r`**×**`r`** blocks in panel L_{21} to abstract processors P_{KL} from the other columns of the processor grid, i.e., abstract processors P_{KL} where $\{I,J,K,L=1\cdots m, J\neq L\}$. This step consists of **`n-(k+1)`** sub-steps. At each such sub-step, the abstract processor P_{IJ} broadcasts a volume of data equivalent to **`(r*r)*sizeof(double)`** to P_{KL}. Since the total volume of data transferred from processor P_{IJ} to processor P_{KL} is **`Get_my_elements(…)*(r*r)*sizeof(double)`**, therefore

  $\frac{100\times r\times r\times sizeof(double)}{Get\_my\_elements(...)\times r\times r\times sizeof(double)}=\frac{100}{Get\_my\_elements(...)}$ percent of all data that should be sent from P_{IJ} to P_{KL} is sent at this step.

- Each abstract processor updates each its **`r`**×**`r`** block of matrix A_{22}. At each such step, the number of **`r`**×**`r`** blocks updated by each abstract processor in A_{22} is given by the function **`Get_my_kk_elements`**, which is illustrated in Figure 5.10(c). Therefore, each abstract processor will perform $\frac{100\times Get\_my\_kk\_elements(...)\times r\times r\times r}{Get\_my\_elements(...)\times n\times r\times r\times r}=\frac{100\times Get\_my\_kk\_elements(...)}{Get\_my\_elements(...)\times n}$ percent of the volume of computations it performs during the execution of the algorithm. The **`par`** algorithmic patterns are used here to specify that all abstract processors perform their computations in parallel.

The performance model **`ParallelCholesky`** shown in the Figure 5.9 is applicable to the heterogeneous algorithm with **`COLUMN_BASED`** data distribution. However this model can be made generic and applicable to any type of data distribution by adding an extra parameter to its parameter list (the type of distribution) and using data partitioning API of HDPI in the body of the performance model. This extra parameter is the type of data distribution such as **`COLUMN_BASED`** or **`ROW_BASED`** or **`CARTESIAN`**.

```
algorithm ParallelCholesky(int m, int r, int n, int l,
         int w[m], int h[m][m][m][m], int type_of_distribution))
```

The most interesting fragments of the rest code of the HMPI parallel application are similar to those shown for HMPI application performing matrix-matrix multiplication in Figure 5.8.

5.3 Experiments with HMPI

This section presents some results of experiments with the HMPI applications presented in Chapters 5.1 and 5.2. Before we present the results, we describe the steps to build and run an HMPI application using Virtual Parallel Machine commands.

5.3.1 Building and Running an HMPI Application

Outlined below are steps to build and run an HMPI application. More details can be obtained from the HMPI Programmer’s guide and installation guide presented in Appendix B.

1). The first step is to describe your Virtual Parallel Machine (VPM). This consists of all the machines being used in your HMPI application. VPM is opened after successful execution of the command **`mpccreate`**. Consider for example the VPM file describing the heterogeneous network shown in Table 5.1:

```
# Machines and the number of processes to run on each machine
# Number in square brackets indicate the number of processors
pg1cluster01 2 [2]
pg1cluster02 2 [2]
pg1cluster03 2 [2]
pg1cluster04 2 [2]
zaphod 1 [1]
csparlx02 1 [1]
csserver 1 [1]
csultra01 1 [1]
csultra02 1 [1]
csultra03 1 [1]
csultra04 1 [1]
cssparc01 1 [1]
```

2). Compile the performance model files.

```
shell$ hmpicc ParallelAxB.mpc
```

This file is translated into a C file “**`ParallelAxB.c`**”.

3). Broadcast the source files to all the machines in the virtual parallel machine.

```
shell$ hmpibcast mxm.c ParallelAxB.c
```

4). Create the target program.

```
shell$ hmpiload –o mxm mxm.c ParallelAxB.c
```

5). Run the target program.

```
shell$ hmpirun mxm
Problem size(n)=1000, time(sec)=13
```

5.3.2 Numerical Results

Note that the figures showing the performances of the computers in the tables in this section give the average speeds measured at runtime during the experiments. The computers used in the experiments are connected to a communication network, which is based on 100 Mbit Ethernet with

Machine Name (Number of Processors)	Architecture	cpu MHz	Total Main Memory (mBytes)	Cache (kBytes)	Relative speed (mxm)	Relative speed (cholesky)
pg1cluster01 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	341
pg1cluster02 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	341
pg1cluster03 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	341
pg1cluster04 (2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	341
zaphod (1)	Linux 2.4.18-14	497	128	512	170	215
csparlx02 (1)	Linux 2.4.18-14	497	256	256	121	170
csserver (1)	Linux 2.4.18-10smp	498	1024	512	105	175
csultra01 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048	46	100
csultra02 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048	46	100
csultra03 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048	46	100
csultra04 (1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048	46	100
cssparc01 (1)	SunOS 5.5 Sun4m sparc SUNW,SPARCstation-5	110	64	512	7	16

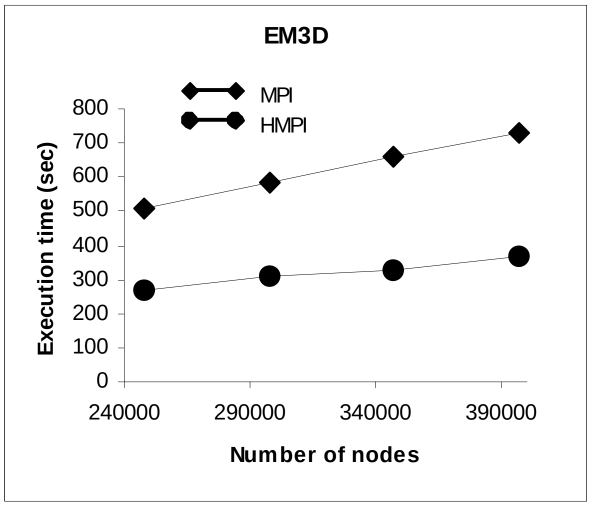
Table 5.1: Specifications of the sixteen heterogeneous processors to demonstrate the efficiency of HMPI over MPI.

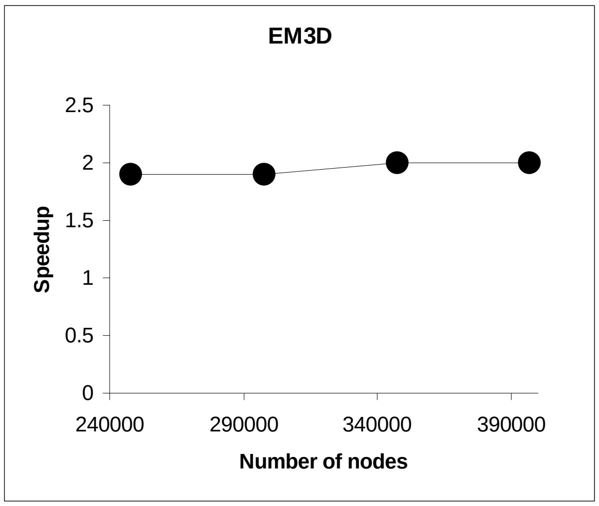
a switch enabling parallel communications between the computers. The experimental results are obtained by averaging the execution times over a number of experiments.

A heterogeneous local network of 16 different Solaris, and Linux workstations shown in Table 5.1 is used in the experiments presented in Figures 5.11, 5.12, and 5.13. The initial static structure of the executing model of this network of computers is automatically obtained by the HMPI environment and saved in the form of an ASCII file as shown below:

```
parallel(0.49, 0.97) c62377 c967039 c801049
#pg1cluster01
s2 p6667 n2 serial c2285064 c107326590 c99523787
```

```
#pg1cluster02

s2 p5556 n2 serial c1312665 c80473880 c98430419

#pg1cluster03

s2 p5556 n2 serial c1956722 c78885862 c99667528

#pg1cluster04

s2 p6905 n2 serial c2540606 c104561354 c100463519

#zaphod

s1 p2632 n1 serial c37569267 c145325990 c292140157

#csparlx02

s1 p2440 n1 serial c17844657 c192059962 c245767189

#csserver

s1 p2223 n1 serial c15790320 c142923824 c213840112

#csultra01

s1 p736 n1 serial c55266123 c197956951 c349384270

#csultra02

s1 p715 n1 serial c44081312 c201423430 c186310535

#csultra03

s1 p720 n1 serial c67108864 c201423430 c295753493

#csultra04

s1 p720 n1 serial c44081312 c201831658 c185115801

#cssparc01

s1 p136 n1 serial c6871230 c39752890 c18018695
```





(a) (b)

Figure 5.11: Results obtained using the heterogeneous network of computers shown in Table 5.1. (a) Comparison of execution times of the EM3D algorithm between HMPI and MPI. (b) The speedup of EM3D algorithm obtained using HMPI over MPI.

Here, each computer is characterized by 7 parameters. The first parameter, **`s`**, determines the number of processors. As can be seen, the computers `pg1cluster01`, `pg1cluster02`, `pg1cluster03`, and `pg1cluster04` are dual processor computers and the rest are uniprocessor computers. The second parameter, **`p`**, is the performance of the computer as determined by the execution of serial test code. In this case the cluster of computers `pg1cluster01`, `pg1cluster02`, `pg1cluster03`, and `pg1cluster04` are the most powerful and the computer `cssparc01` is the least powerful.

Note that at runtime **`HMPI_Recon`** updates this performance value of the parameter for each participating computer. We measure the relative speeds with the core computation of the algorithm (updating of a matrix). Note that the relative speed does not depend on the size of problem for the wide range of matrix sizes used in our experiments. The relative speeds measured for this network are shown in Table 5.1. In the case of matrix-matrix multiplication, the fastest computer `pg1cluster01` is almost 40 times faster than the slowest computer `cssparc01`. In the case of Cholesky factorization, the fastest computer `pg1cluster01` is 20 times faster than the slowest computer `cssparc01`.

The third parameter, **`n`**, determines the total number of parallel processes to run on the computer. In this case one process is run per processor.

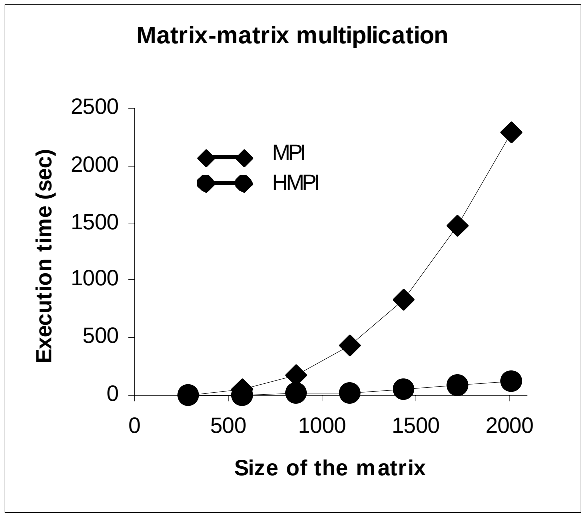
The fourth parameter determines the scalability of the communication layer provided by the computer. In this case, all computers provide serial communication layers.

Finally, the last three parameters determine the speed of point-to-point data transfer between processes running on the same computer as a function of size of the transferred data block size. The first of them specifies the speed of transfer of a data block of 64 bytes (measured in bytes per second), and the second and third are for blocks of 64^2 and 64^3 bytes respectively.

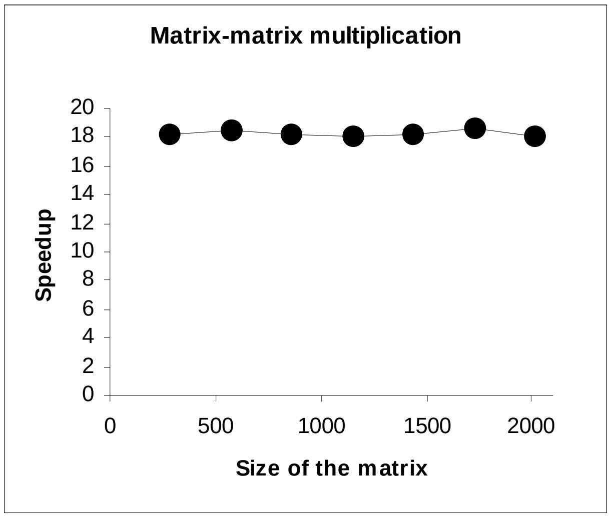
The homogeneous communication space of higher level is also characterized by those three parameters. Besides, the layer is detected as a parallel communication layer with factors 0.49 and 0.97 characterizing the level of parallelism of broadcast and gather correspondingly.

Figure 5.11(a) shows a comparison of the execution times of the HMPI application and the standard MPI application executing the EM3D algorithm. Figure 5.11(b) demonstrates the speedup of the HMPI program over the MPI one. The HMPI application is almost 2 times faster than the standard MPI one.

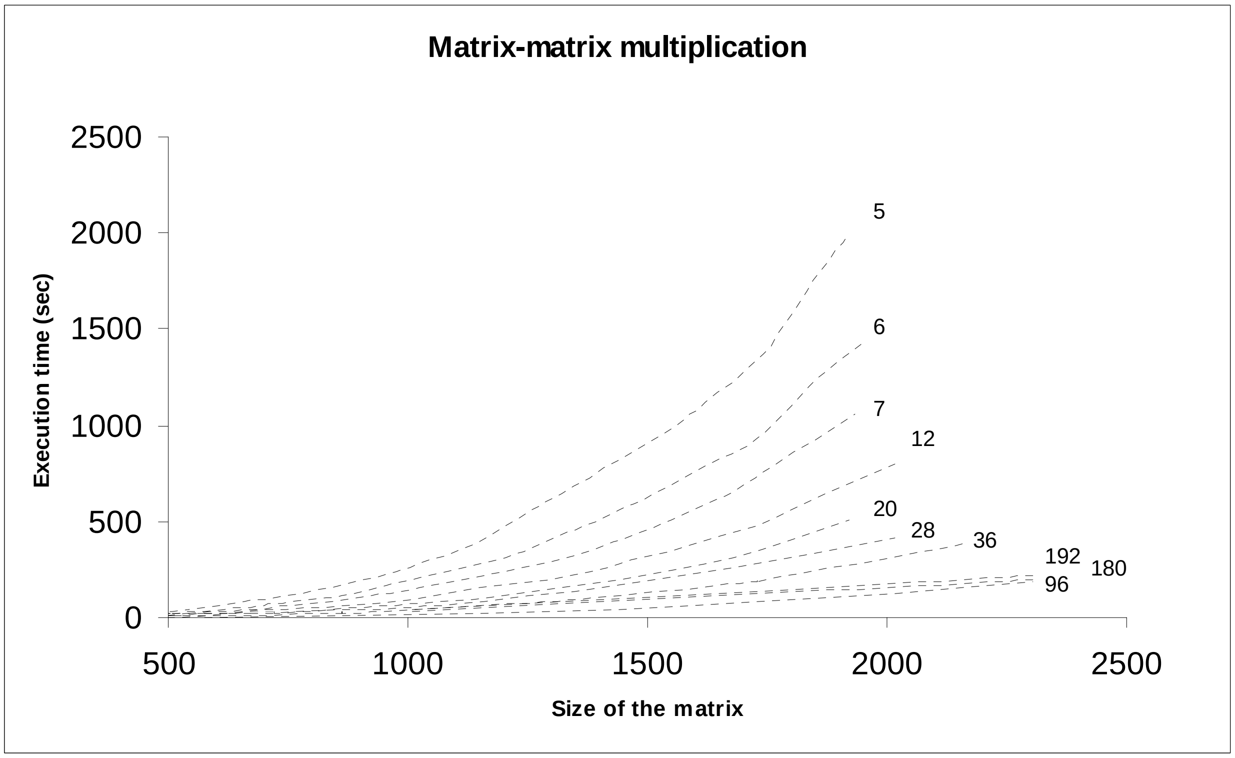
Figure 5.12(a) shows a comparison of the execution times of the MM algorithm between the HMPI application and the standard MPI application using homogeneous 2D block-cyclic data distribution. Figure 5.12(b) demonstrates the speedup of the HMPI program over the MPI one. The results are obtained for the value of **`r`** equal to **8** and the optimal value of the size of generalized block **`l`**, which is shown to be **96** in Figure 5.12(c). It is observed that the execution



(a)

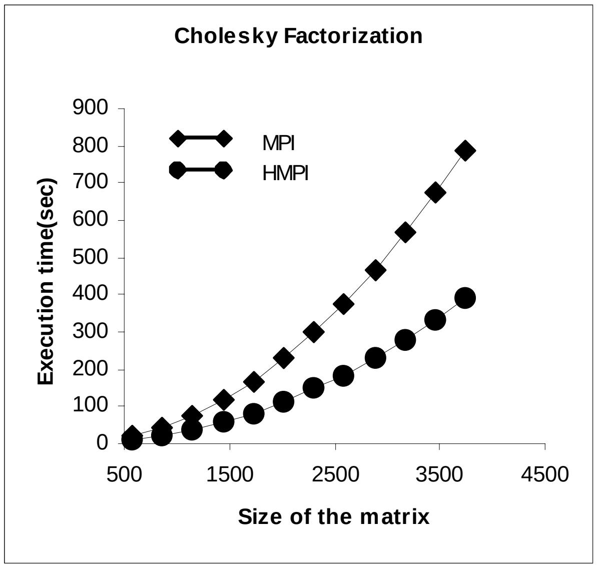


(b)

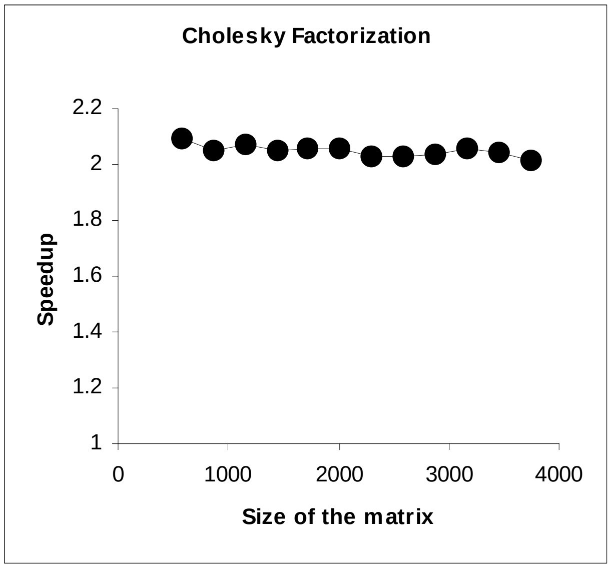


(c)

Figure 5.12: Results obtained using the heterogeneous network of computers shown in Table 5.1. (a) A comparison of the execution times for MM algorithms using HMPI and MPI. (b) The speedup of the MM algorithm obtained using HMPI over MPI. (c) Execution times of the MM algorithm obtained using HMPI for increasing values of generalized block size **l**. The optimal generalized block size is 96.



(a)



(b)

Figure 5.13: Results obtained using the heterogeneous network of computers shown in Table 5.1. (a) A comparison of execution times for the Factorization algorithm using HMPI and MPI. (b) The speedup of the Factorization algorithm obtained using HMPI over MPI.

times of the standard MPI application using homogeneous 2D block-cyclic application are the same no matter what size of generalized block is used, that is, the results are independent of the size of generalized block. Under these circumstances the HMPI application is 18 times faster than the standard MPI one.

Figure 5.13(a) shows a comparison of the execution times of the factorization algorithm between the HMPI application and the standard MPI application using homogeneous 2D block-cyclic data distribution. Figure 5.13(b) demonstrates the speedup of the HMPI program over the MPI one. The results are obtained for the value of `r` equal to **8** and the optimal value of the size of generalized block `l` equal to **72**. The HMPI application is more than 2 times faster than the standard MPI one.

The next set of experimental results presented in Figures 5.14, 5.15, and 5.16 uses a small heterogeneous local network of 9 different FreeBSD, Solaris, and Linux workstations shown in Table 5.2.

The initial static structure of the executing model of this network of computers is automatically obtained by the HMPI environment and saved in the form of ASCII file as shown below:

```
parallel(0.49, 0.97) c209824 c3133982 c10859499
#afflatus
s1 p32001 n1 serial c32311675 c946416696 c299734975
#aries2
s1 p32001 n1 serial c29252581 c423396943 c200739505
#pg1cluster01
s2 p22501 n2 serial c1063941 c58484865 c99137032
#pg1cluster02
s2 p20001 n2 serial c1046987 c59559488 c99244562
#pg1cluster03
s1 p16667 n1 serial c31480669 c613075737 c406484386
#linserver
s1 p16667 n1 serial c19707121 c557314526 c346182102
#csultra01
s1 p4348 n1 serial c54681296 c456174810 c426112471
```

Note that at runtime **`HMPI_Recon`** updates the value of the parameter for each participating computer. We measure the relative speeds with the core computation of the algorithm (updating of a matrix). Note that the relative speed does not depend on the size of problem for the wide

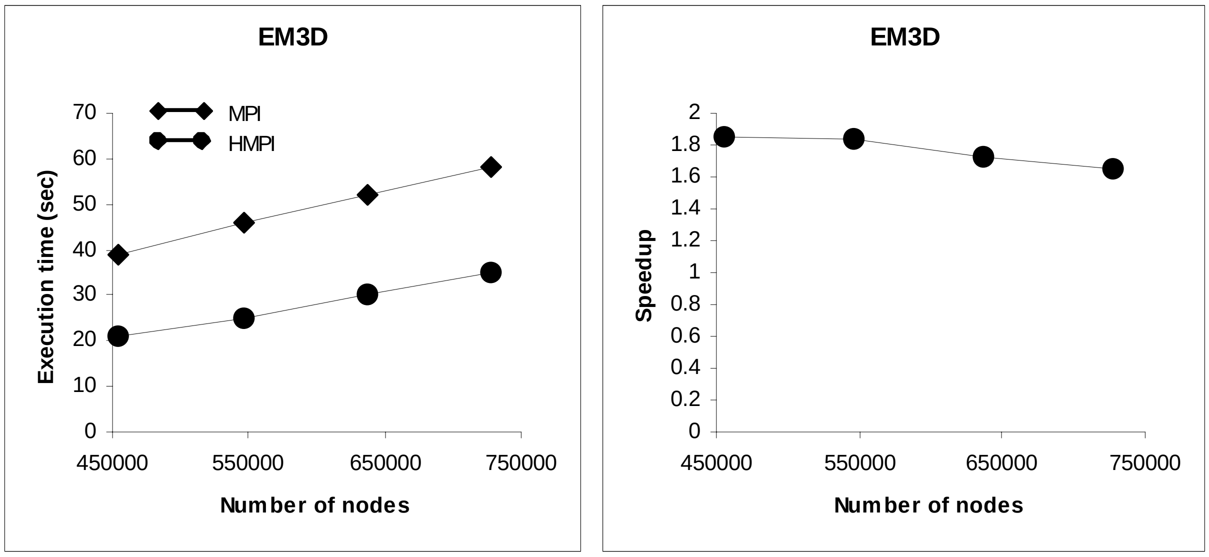
Machine Name (Number of processors)	**Architecture**	**cpu MHz**	**Total Main Memory (mBytes)**	**Cache (kBytes)**	**Relative speed (mxm)**	**Relative speed (cholesky)**
afflatus(1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor supporting HT† technology	2867	2048	1024	499	527
aries2(1)	FreeBSD 5.2.1-RELEASE i386 Intel® Pentium® 4 Processor	2457	512	1024	384	446
pg1cluster01(2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	239
pg1cluster02(2)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	239
pg1cluster03(1)	Linux 2.4.18-10smp Intel(R) XEON(TM)	1977	1024	512	269	239
linserver(1)	Linux 2.4.20-20.9bigmem Intel(R) Xeon(TM)	2783	7748	512	172	351
csultra01(1)	SunOS 5.8 sun4u sparc SUNW,Ultra-5_10	440	512	2048	46	100

Table 5.2: Specifications of the nine heterogeneous processors to demonstrate the efficiency of HMPI over MPI.

range of matrix sizes used in our experiments. The relative speeds measured for this network are shown in Table 5.2. One can see that in the case of matrix-matrix multiplication, the fastest computer `afflatus` is almost 10 times faster than the slowest computer `csultra01`. In the case of Cholesky factorization, the fastest computer `afflatus` is 5 times faster than the slowest computer `csultra01`. So this network is moderately heterogeneous compared to the highly heterogeneous network used in the previous set of experiments. Therefore it is natural to expect the speedup of the heterogeneous HMPI application over the homogeneous MPI application will not be that high.

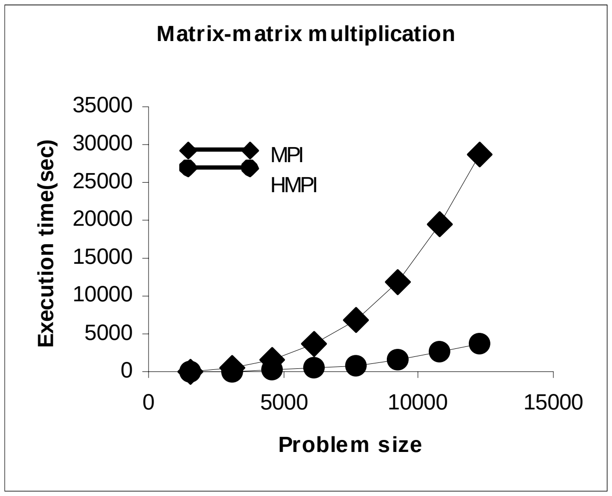
Figure 5.14(a) shows the comparison of the execution times of the HMPI application and the standard MPI application executing the EM3D algorithm. Figure 5.14(b) demonstrates the speedup of the HMPI program over the MPI one. In this case the HMPI application is almost 1.7 times faster than the standard MPI one.

Figure 5.15(a) shows the comparison of the execution times of the MM algorithm between the HMPI application and the standard MPI application using a homogeneous 2D block-cyclic data distribution. Figure 5.15(b) demonstrates the speedup of the HMPI program over MPI. The results are obtained for the value of **r** equal to **16** and the optimal value of the size of generalized block **l,** which is equal to **96**. Using HMPI the application is almost 8 times faster than using standard MPI.

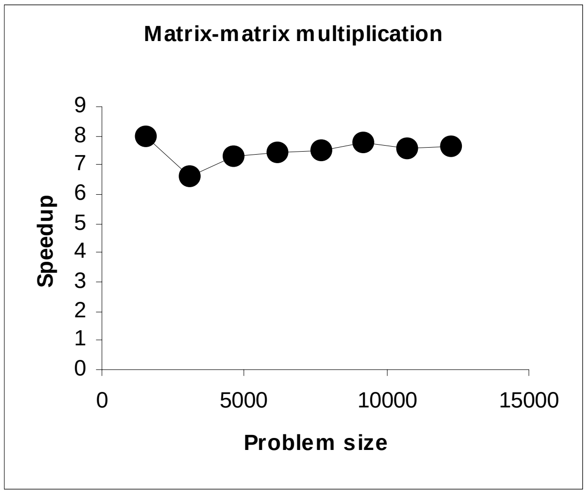


(a) (b)

Figure 5.14: Results obtained using the heterogeneous network of computers shown in Table 5.2. (a) A comparison of execution times of the EM3D algorithm between HMPI and MPI. (b) The speedup of EM3D algorithm obtained using HMPI over MPI.

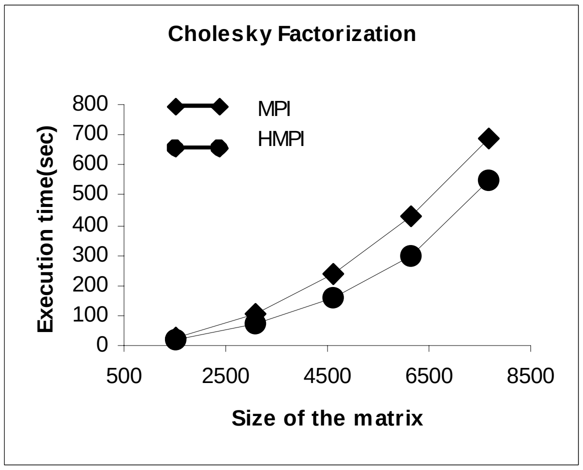


(a)

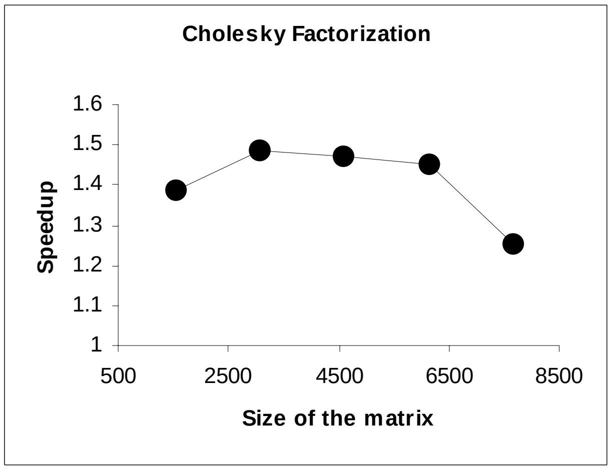


(b)

Figure 5.15: Results obtained using the heterogeneous network of computers shown in Table 5.2. (a) A comparison of execution times of the MM algorithm using HMPI and MPI. (b) The speedup of the MM algorithm obtained using HMPI over MPI.



(a)



(b)

Figure 5.16: Results obtained using the heterogeneous network of computers shown in Table 5.2. (a) A comparison of the execution times of the Factorization algorithm using HMPI and MPI. (b) The speedup of the Factorization algorithm obtained using HMPI over MPI.

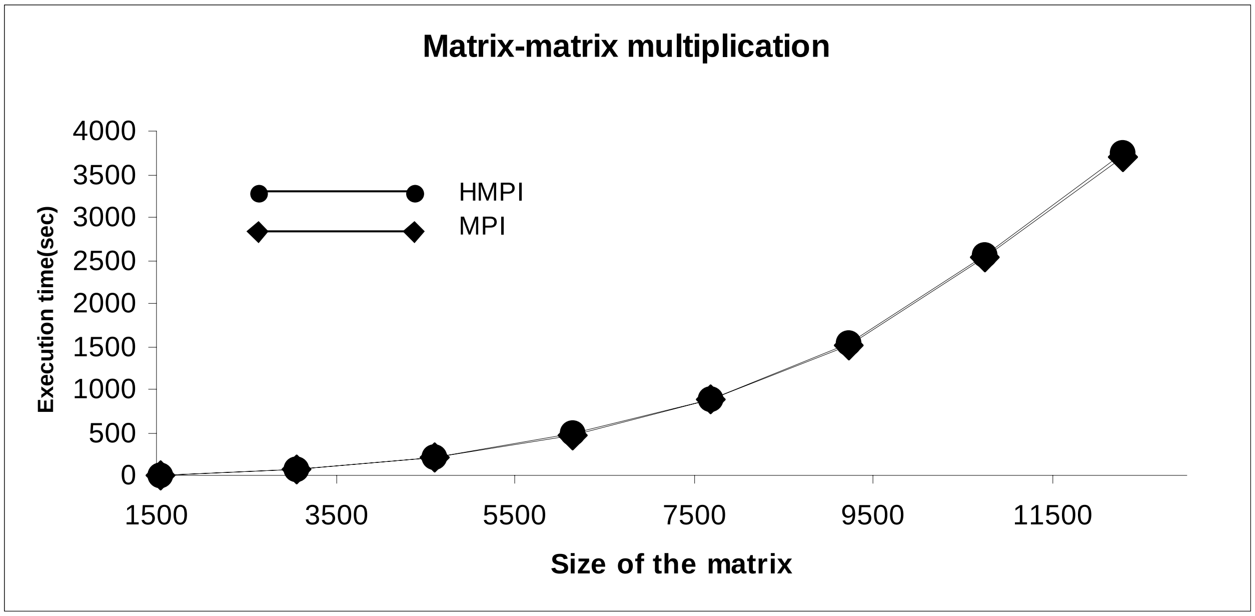


Figure 5.17: Execution times of the HMPI application on the heterogeneous network and the MPI application on the homogeneous network. The two networks have approximately the same aggregate power of processors and share the same (homogeneous) communication network.

Figure 5.16(a) shows a comparison of the execution times of the factorization algorithm between the HMPI application and the standard MPI application using a homogeneous 2D block-cyclic data distribution. Figure 5.16(b) demonstrates the speedup of the HMPI program over the MPI one. The HMPI application is almost 1.5 times faster than the standard MPI one.

Experiments shown in Figure 5.17 compare the efficiency of the HMPI application executing the MM algorithm on a network of nine heterogeneous workstations to the efficiency of the MPI application using homogeneous 2D block-cyclic data distribution executed on a network of 9 identical workstations. The relative speeds of the workstations in the heterogeneous network are 26, 20, 14, 14, 14, 14, 14, 9, and 1. The workstations in the homogeneous network have a same relative speed of 14. The two sets share 5 workstations (of the speed 14) and belong to the same homogeneous communication segment of the local network. The sets were selected so that the aggregate performance of the processors of the heterogeneous network is practically the same as that of the homogeneous one. Thus we compare the efficiency demonstrated by the heterogeneous algorithm on the heterogeneous network with the efficiency demonstrated by its homogeneous prototype on the homogeneous network having the same aggregate performance as the heterogeneous one. From the figure, it can be seen that the applications demonstrate practically the same speed, but each on its network. As the two networks are practically of the same power, we can conclude that the HMPI application cannot perform better and its efficiency is close to optimal on such a heterogeneous network of computers. This approach to analysis of the performance of heterogeneous algorithms is presented in more detail in [AR03, AR04].

5.4 Summary

The experimental results demonstrate that carefully designed HMPI applications can show very good improvements in execution performance on HNOCs. As can be seen, the applications are not fine-tuned for any specific environment. Instead, the performance gains are a result of careful design of applications, which includes:

- Designing an accurate performance model. The performance model definition language of HMPI allows the programmer to describe quite sophisticated heterogeneous parallel algorithms accurately by means of wide use of parameters, locally declared variables, functions, expressions and statements.
- Accurate estimation of performances of the processors using **`HMPI_Recon`**. The accuracy of **`HMPI_Recon`** depends upon how accurately the benchmark code provided by the application programmers reflects the core computations of each phase of their parallel applications. If the benchmark code provided is an accurate measurement of the core computations in each phase, **`HMPI_Recon`** gives an accurate measure of the speeds.

- The accuracy of **`HMPI_Timeof`** in finding the optimal values of the parameters of the parallel algorithm. This in turn depends on the accuracy of the performance model designed, the quality of the heuristics designed for the set of parameters provided to the performance model, and the accuracy of the model of the executing network of computers.

From the performance models presented in this chapter, it can be seen that a performance model can be written that is generic enough to be used for any type of data distribution. The generality of the performance model is achieved through using generic parameters in its parameter list and using data partitioning API of HDPI in the body of the performance model. Such performance models are only written once and used for different types of data distribution.

HMPI applications once developed and that follow each of the steps outlined below will run efficiently on any HNOCs without any changes to its source code (we call the property *efficient portability*):

- Initialization of the HMPI runtime system with **`HMPI_Init`**;
- Estimation of the speeds of processors with **`HMPI_Recon`**;
- Finding the optimal values of the parameters of the parallel algorithm with **`HMPI_Timeof`**;
- Creation of a group of processes, which will perform the parallel algorithm, by using **`HMPI_Group_create`** or using **`HMPI_Group_auto_create`**.
- Execution of the computations and communications of the heterogeneous parallel algorithm by the members of the group using standard MPI means. At this step, application programmers use the MPI communicator with communication group of MPI

processes given by the handle to HMPI group of processes to call the standard MPI communication routines.

- Freeing the HMPI groups with **`HMPI_Finalize`**;
- Finalization of the HMPI runtime system with **`HMPI_Init`**;

On the whole, the experiments demonstrate that the HMPI provides all the features to guide the user to write portable and efficient applications on HNOCs.

CHAPTER 6

Conclusions

HMPI offers a unifying framework designed specially for programming high-performance computations on HNOCs. HMPI provides all the features to the user to write portable and efficient parallel applications on HNOCs. These features automate all the essential steps involved in application development on HNOCs:

1). Determination of the characterization parameters relevant to the computational requirements of the applications and the machine capabilities of the heterogeneous system. These parameters are determined before the application execution and form the model of executing network of computers. The representation of the model of the executing network of computers is implementation-dependent. In our research implementation of HMPI, during the creation of a virtual parallel machine, a static structure that represents the model is generated and saved in the form of an ASCII file. HMPI provides interfaces to update the parameters of the model at runtime taking into account the fluctuations of the network load.

2). Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes. This step of heterogeneous decomposition is parameterized by the characterization parameters determined in the first step, mainly, the speeds of processors and the latencies and bandwidths of the communication links between them, and the user-available memory capacity of the machine. The Heterogeneous Data Partitioning Interface (HDPI) is developed to automate this step of heterogeneous decomposition. HDPI provides API that allows the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition the mathematical objects used in their parallel applications.

3). Selection of the optimal set of processes running on different computers of the heterogeneous network by taking into account the speeds of the processors, and the latencies and the bandwidths of the communications links between them. HMPI provides a small set of extensions to MPI, which automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group. During the creation of this set of optimal processes, HMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network using an advanced mapping algorithm. The mapping algorithm is based on the performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

4). Application program execution on the HNOCs. The command line user interface of HMPI developed consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of the HMPI application programs on the virtual parallel machine. The notion of virtual parallel machine enables a collection of heterogeneous computers to be used as single large parallel computer.

The merits of HMPI were demonstrated through the design, analysis, and implementation of three applications on HNOCs. They are Matrix-matrix multiplication, Cholesky Factorization, and EM3D. These applications are representative of many scientific applications. Experimental results show that carefully designed HMPI applications can show very good improvements in execution performance on HNOCs.

Once developed, an HMPI application will run efficiently on any HNOCs without any changes to its source code (we call the property *efficient portability*). It can be seen that the improved performance of the HMPI applications is not due to the fine-tuning of these applications to a specific environment. By hiding the non-uniformity of the underlying heterogeneous system from the application programmer, the HMPI offers an environment that encourages the design of heterogeneous parallel software in an architecture-independent manner.

6.1 Contributions

Below, we present more precisely the contributions of this work.

a) The design of HMPI API, which are extensions to MPI, to automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group. The main goal of the design of the API in HMPI is to smoothly and naturally extend the MPI model for heterogeneous networks of computers. This involves the design of a layer above MPI that does not involve any changes to the existing MPI API. The HMPI API must be easy-to-use and suitable for most scientific applications. The HMPI API must also facilitate transformation of MPI applications to HMPI applications that run efficiently on HNOCs.

b) The first research implementation of HMPI.

c) The design and application of HMPI+ScaLAPACK tool to speed up ScaLAPACK applications on HNOCs.

d) Design and implementation of Heterogeneous Data Partitioning Interface (HPDI) to automate the step of heterogeneous decomposition in the solution of parallel problems on HNOCs.

e) The design of efficient set partitioning algorithms using a realistic performance model of networks of heterogeneous computers. These algorithms solve the problem of optimal distribution of computational tasks on a network of heterogeneous computers when one or more tasks do not fit into the main memory of the processors and when relative speeds cannot be accurately approximated by constant functions of problem size.

f) HMPI application programming illustrating the usage of performance models, API of HMPI, and the API of HPDI, and experimental results demonstrating efficient, scalable, and predictable HMPI applications.

HMPI is a small set of extensions to MPI, which facilitate the writing of parallel programs that distribute computations and communications unevenly, taking into account the speeds of the processors, and the latencies and bandwidths of communication links. The main goal of the design of API in HMPI is to smoothly and naturally extend MPI model for heterogeneous networks of computers. This involves the design of a layer above MPI that does not involve any changes to the existing MPI API. The HMPI API must be easy-to-use and suitable for most scientific applications.

The main idea of HMPI is to automate the process of selection of such a group of processes that executes the heterogeneous algorithm faster than any other group. HMPI allows the application programmers to describe a performance model of their implemented heterogeneous algorithm. This model allows for all the main features of the underlying parallel algorithm that have an essential impact on application execution performance on HNOCs.

HMPI provides a small and dedicated model definition language for specifying this performance model. A compiler compiles the description of this performance model to generate a set of functions. The functions make up an algorithm-specific part of the HMPI runtime system.

Having provided such a description of the performance model, application programmers can use the HMPI group constructor functions, which try to create a group that would execute the heterogeneous algorithm faster than any other group of processes. During the creation of this group of processes, HMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network using an advanced mapping algorithm. The mapping algorithm is based on the performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

HMPI also provides interfaces that allow application programmers to write applications adapting not only to nominal performances of processors but also to redistribute computations and communications dependent on dynamic changes of workload of separate computers of the executing network.

The implementation of Heterogeneous Data Partitioning Interface (HPDI) is a C library of routines that allow the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition the mathematical objects used in their parallel applications. The design of HPDI is based on a classification of partitioning problems on networks of heterogeneous computers. Our approach to classification of partitioning problems is based on two corner stones:

- A realistic performance model of networks of heterogeneous computers,
- A natural classification of mathematical objects most commonly used in scientific, engineering and business domains for parallel (and distributed) solving problems on networks of heterogeneous computers.

HPDI is designed to be used along with various programming tools for parallel and distributed computing on heterogeneous networks.

To demonstrate the efficiency of HPDI, we perform experiments using naïve parallel algorithms for linear algebra kernel, namely, matrix multiplication and Cholesky factorization using striped partitioning of matrices on a local network of heterogeneous computers. Our main aim is not to show how matrices can be efficiently multiplied or efficiently factorized but to explain in simple terms the usage of this API. We also view these algorithms as good representatives of a large class of data parallel computational problems and a good testing platform before experimenting more challenging computational problems. The results show good improvement in performance on networks of heterogeneous computers.

To investigate the merits of HMPI, we designed, analyzed, and implemented three applications on HNOCs, namely, Matrix-matrix multiplication, Cholesky Factorization, and EM3D. These applications are representative of many scientific applications. Experimental results show that carefully designed HMPI applications can show very good improvements in execution performance on HNOCs. These applications follow each of the steps outlined below:

- Initialization of the HMPI runtime system with **`HMPI_Init`**;
- Estimation of the speeds of processors with **`HMPI_Recon`**;
- Finding the optimal values of the parameters of the parallel algorithm with **`HMPI_Timeof`**;
- Creation of a group of processes, which will perform the parallel algorithm, by using **`HMPI_Group_create`** or using **`HMPI_Group_auto_create`**.
- Execution of the computations and communications of the heterogeneous parallel algorithm by the members of the group using standard MPI means. At this step, application programmers use the MPI communicator with communication group of MPI processes given by the handle to HMPI group of processes to call the standard MPI communication routines.
- Freeing the HMPI groups with **`HMPI_Finalize`**;
- Finalization of the HMPI runtime system with **`HMPI_Init`**;

It can be seen that these applications can run efficiently on any HNOCs without any changes to its source code. That is, these applications are not fine-tuned for any specific environment. Instead, the performance gains are a result of careful design, which includes:

- Designing an accurate performance model.
- Accurate estimation of performances of the processors using **`HMPI_Recon`**.
- Finding the optimal values of the parameters of the parallel algorithm using **`HMPI_Timeof`**. The high accuracy of the estimation by **`HMPI_Timeof`** is ensured

because for each of these applications, an accurate performance model was designed, high quality heuristics were designed for the set of parameters provided to the performance model, and the model of the executing network of computers used was accurate enough.

From the performance models presented, it can be seen that a performance model can be written that is generic enough to be used for any type of data distribution. The generality of the performance model is achieved through using generic parameters in its parameter list and using data partitioning API of the HDPI in the body of the performance model. Such performance models are only written once and used for different types of data distribution.

CHAPTER 7

Future Work

Based on the lessons learned from the development, implementation, and evaluation of HMPI, the following research extensions are presented as future efforts:

- *Develop a fault-tolerant HMPI implementation.* Currently, HMPI does not provide any means for the writing of fault-tolerant parallel applications for HNOCs. The implementation of FT-HMPI could be based on the following approaches to writing fault-tolerant programs.
    - *Checkpointing.* Checkpointing is a common technique that periodically saves the state of a computation, allowing the computation to be restarted from that point in the event of failure. Checkpointing is easy to implement but is often considered expensive. For small probability of failure and relatively small costs of creating and restoring a checkpoint, the added cost of using checkpoints is quite small. Since checkpoints must be saved to persistent storage that is not affected by a failure of one of the computing elements, the checkpoint data are typically saved to a (parallel) file system. Thus, the practicality of checkpointing is related to the performance of parallel I/O. MPI provides excellent facilities for performing I/O.

      There are two types of checkpointing: *user-directed* and *system-directed*. In user-directed checkpointing, the application programmer forms the checkpoint, writing out any data that will be needed to restart the application. This task is often relatively easy, particularly with well-structured applications. Unfortunately, system-directed checkpointing is much harder to implement because so much of the process's state is scattered throughout a parallel computer. This state can include messages that are in flight between processes and data in kernel memory buffers.

For user-directed checkpointing, source code transformation tools based on compiler technology can be used to identify both what data to checkpoint and what data need not be saved in a particular checkpoint.

- *Using intercommunicators.* In MPI, communicators are distributed objects that are used for collective and point-to-point operations. Because of collective operations, the failure of any one process in a communicator affects all processes in the communicator, even those that are not in direct communication with the failed process. In contrast, in non-MPI client-server programs, the failure of the client has no effect on the server. This structure is robust because all communication takes place in a two-party context, in which one party can recognise that other party has failed and cease communicating with it. MPI programs can be structured to have this same type of survivability by using "intercommunicators", which consists of two groups of processes, and all communication occurs between processes in one group and processes in the other group.
- *Modifying MPI semantics.* This approach to fault tolerance modifies the semantics of certain MPI objects and functions. FT-MPI [FBD01] is a research implementation of MPI that allows the semantics and associated modes of failures to be explicitly controlled by an application via a modified MPI API. FT-MPI provides application programmers with different methods of dealing with failures within MPI application than just checkpoint and restart. It allows the application to continue using a

communicator with a failed rank while explicitly excluding communication with the failed rank, or to shrink the communicator by excluding the failed rank, or to spawn a new process to take the place of a failed process.

  - *Extending MPI.* Rather than modifying existing MPI semantics, extensions to MPI can be defined that have semantics that support the writing of fault-tolerant programs but are all consistent with all existing MPI semantics. The key idea here is to encapsulate the capabilities using intercommunicators, where instead of using **`MPI_COMM_WORLD`**, communication is based on a local array of two-party connections. To add new capabilities for expressing fault-tolerant constructs in an MPI context, new MPI objects and methods need be designed.

- *Develop HMPI for High Performance Grid Computing*. HMPI must be extended to provide means to facilitate writing of high performance grid computing applications. The main tasks involve:
  - Development of an advanced performance model of the Grid as a heterogeneous environment of distributed computational resources;
  - Design and implementation of a system component that will discover and maintain the parameters of the performance Grid model;
  - Design and implementation of algorithms of the optimal mapping of the Grid application to available computational resources;
  - Design and implementation of the compiler translating the specification of the Grid application into a program component that dynamically maps the application to the available global computing resources;

Currently HMPI provides utilities automating all the above steps but these are geared towards HNOCs. They must be extended to work for computing grids.

- *Design of a Heterogeneous Linear Algebra Package (HLAPACK).* The HLAPACK library will include a subset of LAPACK routines redesigned for HNOCs. Like LAPACK, the HLAPACK routines will be based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of memory hierarchy. The fundamental building blocks of the HetLAPACK library will be distributed-memory versions of the Level 1, Level 2, and Level 3 BLAS, called the Parallel BLAS or PBLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. The design of HLAPACK will include:
    - Design of performance models for each of the level-1, level-2, and level-3 BLAS routines.
    - Design of API for level-1, level-2, and level-3 BLAS routines. The interface will look similar to LAPACK. Each of these routines will create a group of processes with an optimal number of processes. This is followed by the execution of the heterogeneous algorithm associated with these routines by the members of the group.
- *Improvements to the Heterogeneous Data Partitioning Interface (HPDI).* We presented the Heterogeneous Data Partitioning Interface (HPDI), which provides API for partitioning mathematical objects commonly used in scientific and engineering domains for solving problems on networks of heterogeneous computers. However, more work needs to be done to further improve this library, which includes:

  - Mathematical analysis of functions representing the absolute speed of a processor followed by the classification of the shapes of the speed function. This is followed by development of procedures for building speed functions, algorithms of calculation of shape of speed functions, and software for calculation of shapes of speed functions.
  - Classification of data partitioning problems for dense matrices involved in $O(n^2)$ algorithms, $O(n^3)$ algorithms, and $O(n^r)$ algorithms where $r>3$. This is followed by design and collection of data partitioning algorithms for dense matrices involved in $O(n^2)$ algorithms, $O(n^3)$ algorithms, and $O(n^r)$ algorithms where $r>3$. Then API would have to be designed for the algorithms for dense matrices involved in $O(n^2)$ algorithms, $O(n^3)$ algorithms, and $O(n^r)$ algorithms where $r>3$.
  - Classification of data partitioning problems for multidimensional arrangements. This is followed by design and collection of data partitioning algorithms for multidimensional arrangements and design of API to the algorithms for multidimensional arrangements.
  - Classification of data partitioning problems for graphs and trees. This is followed by design and collection of data partitioning algorithms and design of API to the algorithms for graphs and trees.

- *Extension of set partitioning algorithms.* Representation of the dependence of the speed on the problem size by a single curve is reasonable for computers with moderate fluctuations in workload because in this case the width of the performance band is quite narrow. On networks with significant workload fluctuations, the speed function of the problem size should be characterized by a band of curves rather than by a single curve.

We aim to design efficient algorithms of data partitioning on heterogeneous networks of computers where the speed of a processor is represented by a speed band, the width of the band characterizing fluctuations in speed due to changes in load over time.

The set partitioning algorithms presented in this work assume that the volume of computations involved in the execution of the problem size assigned to a processor is proportional to the problem size. That is the volume of computations is proportional to the number of elements in the partition assigned to the processor. We aim to extend the set partitioning algorithms for applications where the volume of computations involved in the execution of a problem size is not proportional to the problem size. In some such cases, the functional performance model where the speed of the processor is represented by a smooth continuous function of problem size can be creatively modified and our set partitioning algorithms can be applied.

- *Incorporation of communication cost into the functional performance model of HNOCs.*

  We have proposed an functional model of a network of heterogeneous computers and designed efficient algorithms of data partitioning with this model. Under this model, the speed of each processor is represented by a continuous and relatively smooth function of the problem size. This model integrates the effects of paging. However, the other essential features underlying applications run on networks of heterogeneous computers such as the latency and the bandwidth of the communication links between the processors are not considered. These parameters should be incorporated into the model to make it more comprehensive and efficient data partitioning algorithms need to be developed using this model.

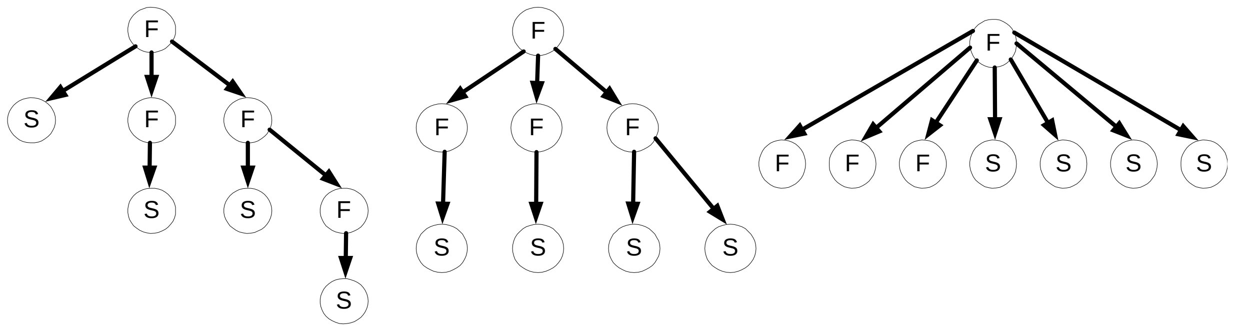
- *Improvements to the communication model of executing network of computers.* In HMPI, the model of executing networks of computers treats any set of parallel communications as if they all take place at the same communication layer in the hierarchy, namely, at the lowest communication layer covering all involved processors. In reality, some of the communications may use different communication layers. Incorporation of multi-layer parallel communications in this algorithm without significant loss of its efficiency is a very difficult problem, which needs to be addressed.

  The model of executing network of computers in HMPI uses three parameters that determine the speed of point-to-point data transfer between processes running on the same computer as function of size of the transferred data block. The speed of the transfer of a data block of an arbitrary size is calculated by interpolation of the measured speeds. This model is efficient but not very accurate. The model needs to be improved with inclusion of parameters, which model point-to-point and collective communications accurately and efficiently:

  - The time taken for a point-to-point communication involving processes running on the same computer can be represented by linear dependence on the message size. Parameters must be designed to represent the fixed and the variable components.
  - The cost of a point-to-point message transfer involving processes running on different computers usually consists of three components, namely, the send overhead, transmission cost, and the receive overhead. Suitable parameters must be designed to accurately estimate these fixed and variable components.

- Collective communications such as broadcast, gather, and scatter can involve processes running on the same or different computers. In case of collective communications involving processes running on the same computer, it is observed that the time taken for the collective communication can be approximated by a constant linear function of the number of nodes involved in the collective communications. Thus the parameter that needs to be included for these types of communications represents the constant in the linear function. For collective communications involving processes running on different computers, the cost model used to estimate the time taken for the collective communication must take into account the collective communication algorithm used. On HNOCs, collective communication operations can be implemented by using different algorithms, which use different types of trees. The collective communication algorithm must take into account the communication capabilities of the participating nodes. This algorithm must ensure that slow computers are involved in less number of communications than the fast computers. This can be ensured by making the slow computers occupy the leaf nodes of the tree and faster computers occupy the intermediate nodes to broadcast the message faster. Thus parameters must be determined to incorporate the different types of schemes used by application programmers for their collective communication algorithms.

The issue of determination of the parameters to accurately and efficiently model these different types of communications is difficult, which needs to be addressed. The model of executing network of computers must also take into account the contention that may be



(a) Binomial Tree (b) Hierarchical Tree (c) Sequential tree

Figure 7.1: Three different trees for implementing broadcast in a Heterogeneous network of computers with eight nodes (F=fast, S=slow).

caused in the network. Already models such as LoGPC [MF98, FB96] exist that account for the impact of network contention effects on the performance of message-passing programs.

- *Improvements to the communication pattern specification in the scheme declaration of the performance model definition language.* The rule for estimation of the execution time of the parallel algorithmic pattern in the performance model definition language is the core of the entire mapping algorithm determining its accuracy and efficiency. Most disadvantages of the rule are just the backside of its simplicity and the necessity to keep it effective. Except some common collective communication operations, it is not sensitive to different collective communication patterns such as ring data shifting, tree reduction, etc., treating all of them as a set of independent point-to-point communications. The main problem is that recognition of such patterns is very expensive. A possible solution is introduction in the performance model definition language some explicit constructs for communication pattern specification as a part of the scheme description.

- *Design of collective communication routines for HNOCs.* On HNOCs, collective communication routines can be implemented by using different algorithms, which use different types of trees as shown in Figure 7.1. The most commonly used collective communication routines are broadcast, gather, scatter, and barrier synchronization. It is observed that on HNOCs, the collective communication algorithm used must take into account the communication capabilities of the participating nodes. This algorithm must ensure that slow computers are involved in less number of communications than the fast computers. This can be ensured by making the slow computers occupy the leaf nodes of the tree and faster computers occupy the intermediate nodes to broadcast the message faster. Therefore, collective communication routines must be provided to the application programmers that efficiently perform the collective communications. Each of the routines must use the communication cost model to compare the performance of different schemes in order to find the best scheme for a given collective operation.
- *Design of interfaces that update the communication parameters at runtime to be used by the performance measurement models of HMPI runtime system.* The main idea of HMPI is to automate the process of selection of a group of processes, which would execute the heterogeneous algorithm faster than any other group. One of the features that affect the efficiency of the process of selection is the accuracy of the model of the executing network of computers. This depends on the accuracy of the measurements of the processor speeds given by **`HMPI_Recon`** and the communication model of the executing network of computers. Currently the communication model used in HMPI runtime

system is static. The issue of efficiently updating the parameters of communication model at runtime needs to be addressed.

- *Tool for converting MPI programs to HMPI programs.* A tool needs to be developed that would automatically make some straightforward transformations to convert an MPI program to a HMPI program. The tool could be as simple as a script or a preprocessor that generates a basic working version of an HMPI program from an input MPI program. All that the application programmer will have to do is to design a performance model and input this performance model and MPI programs to the compiler or preprocessor. Hooks can be provided that allow the application programmers to specify the different stages of an MPI program that would aid the transformation process. These are the following:
  - MPI initialization,
  - Data distribution,
  - Execution of the algorithm by the processes of **`MPI_COMM_WORLD`**, and
  - MPI finalization.

  Based on this information, a basic working version of a HMPI program can be generated from the performance model provided by the application programmer and the static program analysis of the MPI program. The basic working version would contain the following:
  - HMPI initialization replacing the MPI initialization,
  - Data distribution using the speeds of the processors. This step uses the API of the Heterogeneous Data Partitioning Interface (HDPI). The application programmer must

dynamically update the processor speeds at runtime using **`HMPI_Recon`** before distributing the data.

- Creation of a HMPI group of processes using HMPI group constructor functions (**`HMPI_Group_create`** or **`HMPI_Group_auto_create`**). The handle to the performance model in the group creation function is generated by compiling the performance model provided as input by the application programmer. The application programmer will have to fill in the model parameters using the function **`HMPI_Pack_model_parameters`**.
- Execution of the algorithm by the processes of MPI communicator associated with the HMPI group of processes. This piece of code is similar to the MPI code except that the MPI communicator **`MPI_COMM_WORLD`** is replaced by the MPI communicator associated with the HMPI group of processes.
- Destruction of the HMPI group of processes. The call to the group destruction function **`HMPI_Group_free`** is inserted, and
- HMPI finalization replacing the MPI finalization.

- *Design additional HMPI applications.* HMPI needs to be experimented on a wide variety of real-life scientific applications and on wide variety of HNOCs to improve the specification of the performance model definition language and the model of executing network of computers. Such experiments will also help investigate and strengthen the property of *efficient portability* of HMPI applications. That is, once developed, an HMPI application will run efficiently on any HNOCs without any changes to its source code.

- *Bindings of HMPI to C++, FORTRAN, and Java.* HMPI API has currently bindings to ANSI C. Convenient bindings to C++, FORTRAN, and Java must be developed to attract programmers who use one or more of these languages to write message-passing programs on HNOCs.

List of References

[AAL+95] S. Amarasinghe, J. Anderson, M. Lam, and C. Tseng, "The SUIF compiler for scalable parallel machines," In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[AAB+00] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, "An opportunity cost approach for job assignment in a scalable computing cluster," In *IEEE Transactions on Parallel and Distributed Systems*, Volume 11, No. 7, pp.760-768, July 2000.

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. LAPACK Users' Guide, Release 1.0, *SIAM*, Philadelphia, 1992.

[ABN00] O. Aumage, L. Bouge, and R. Namyst, "A Portable and Adaptative Multi-Protocol Communication Library for Multithreaded Runtime Systems," In Proceedings of the fourth Workshop on runtime systems for Parallel *Programming (RTSPP'00)*, *Lecture Notes in Computer Science,* Volume 1800, Springer Verlag, Cancun, Mexico, 2000, pp.1136-1143.

[ACP95] T. E. Anderson, D. E. Culler, and D. A. Patterson, "A Case for NOW (Networks of Workstations)," In *IEEE Micro*, February 1995.

[AIS+95] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation," In *7th Annual Symposium on Parallel Algorithms and Architecture (SPAA'95)*, July 1995.

[AKL+99] D. Arapov, A. Kalinov, A. Lastovetsky, and I. Ledovskih, "A Language Approach to High Performance Computing on Heterogeneous Networks," In *Parallel and Distributed Computing Practices*, Volume 2, No. 3, pp.87-96, 1999.

[AMJ+03] J. Al-Jaroodi, N. Mohamed, H. Jiang, and D. Swanson, "Modeling Parallel Applications Performance On Heterogeneous Systems", In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, *Workshop on Advances in Parallel and Distributed Computational Models*, 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2003.

[Ang98] C. Anglano, "Predicting Parallel Applications Performance on Non-Dedicated Cluster Platforms," In *International Conference on Supercomputing*, 1998, pp.172-179.

[AR03] A. Lastovetsky, and R. Reddy, An Approach to Assessment of Heterogeneous Parallel Algorithms, In *The 7th International Conference* *on Parallel Computing Technologies (PaCT 2003)*, Nizhni Novgorod, Russia, Lecture Notes in Computer Science, Volume 2763, 2003, pp.117-129.

[AR04] A. Lastovetsky and R. Reddy, "On Performance Analysis of Heterogeneous Parallel Algorithms," In *Parallel Computing*, Volume 30, No. 11, pp.1195-1216, 2004.

[ASW+98] C. A. Lee, J. Stepanek, R. Wolski, C. Kesselman, and I. Foster, "A Network Performance Tool for Grid Environments", *In Proceedings of seventh IEEE International Symposium on High Performance Distributed Computing*, 1998, pp.260-267.

[BBL+00] O. Beaumont, V. Boudet, A. Legrand, F. Rastello, and Y. Robert. Heterogeneity considered harmful to algorithm designers. *Technical Report RR-2000-24*, LIP, ENS Lyon, June 2000.

[BBR+00] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Partitioning a square into rectangles: NP-completeness and approximation algorithms. *Technical Report RR-2000-10*, LIP, ENS Lyon, February 2000.

[BBR+01] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix Multiplication on Heterogeneous Platforms," In *IEEE Transactions on Parallel and Distributed Systems*, Volume 12, No. 10, pp.1033-1051, October 2001.

[BBP+01] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers)," In *IEEE Transactions on Computers*, Volume 50, No. 10, pp.1052-1070, October 2001.

[BCC+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, ScaLAPACK Users' Guide. In *SIAM*, 1997.

[BDV94] G. Burns, R. Daoud, and J. Vaigl, "LAM: An open cluster environment for MPI," In *Proceedings of Supercomputing Symposium '94*, University of Toronto, Toronto, 1994, pp. 379-386.

[BFR03] R. Baraglia, R. Ferrini, and P. Ritrovato. A Static Mapping Heuristic for Mapping Parallel Applications to Heterogeneous Computing Systems. *Technical report 2003-TR-08*, Institute of Science and Technologies, University of the Studies of Salerno, 2003.

[BGM+96] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press and John Wiley & Sons, 312 pages, August 1996.

[BGM+97] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," Modern Software Tools in Scientific Computing, E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, pp.163-202, 1997.

[BJ93] T. Bui and C. Jones, "A heuristic for reducing fill in sparse matrix factorization," In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 1993, pp.445-452.

[BKS+01] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger, "Object-Based Adaptive Load Balancing for MPI Programs," In *Proceedings of the International Conference on Computational Science*, San Francisco, CA, Lecture Notes in Computer Science, Volume 2074, Springer Verlag, May 2001, pp.108-117.

[BMD98] M. Banikazemi, V. Moorthy, and D. Panda, "Efficient Collective Communication on Heterogeneous Networks of Workstations," In *International Conference on Parallel Processing,* Minneapolis, MN, 1998, pp. 460-467.

[BMP04] J. Barbosa, C. N. Morais, and A. J. Padilha, "Simulation of Data Distribution Strategies for LU Factorization on Heterogeneous Machines," In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 26-30 April 2004, New Mexico, France, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2004.

[BN97] G. Blelloch and G. Narlikar, "A Practical Comparison of N-Body Algorithms," In *Parallel Algorithms*, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* American Mathematical Society, Volume 30, pp.81-96, 1997.

[BNC+01] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. S. Dandass, and M. Apte, "MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing," In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001.

[BPB+99] L. Birov, A. Prokofiev, Y. Bartenev, A. Vargin, A. Purkayastha, Y. Dandass, V. Erzunov, E. Shanikova, A. Skjellum, P. Bangalore, E. Shuvalov, V. Ovechkin, N. Frolova, S. Orlov, S. Egorov, "The Parallel Mathematical Libraries Project (PMLP): Overview, Design Innovations, and Preliminary Results," In *Proceedings of the 5th International Conference on Parallel Computing Technologies*, pp.186-193, September 06-10, 1999.

[BPR99] P. Bhat, V. K. Prasanna, and C. S. Raghavendra, "Block-Cyclic Redistribution over Heterogeneous Networks," In *Proceedings of the Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 1999),* Fort Lauderdale, Florida, August 18-20, 1999.

[BPS82] R. I. Becker, S. R. Schach, and Y. Perl, "A Shifting Algorithm for Min-Max Tree

Partitioning," In *Journal of the ACM*, Volume 29, No.1, pp.58-67, January 1982.

[BRP99] P. Bhat, C.S. Raghavendra, and V. Prasanna, "Efficient Collective Communication in Distributed Heterogeneous Systems," In *International Conference on Distributed Computing Systems*, May 1999.

[BS89] S. T. Barnard and H. D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems" In *Proceedings of the 3rd International Conference on Genetic Algorithms,* George Mason University, Fairfax, Virginia, USA, June 1989, pp.116-121.

[BSB+99] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison study of static mapping heuristics for a class of meta-tasks on heterogeneous computing systems," In *8th IEEE Workshop on Heterogeneous Computing Systems (HCW '99),* San Juan, Puerto Rico, April 1999, pp.15-29.

[BTP00] J. Barbosa, J. Tavares, and A. J. Padilha, "Linear Algebra Algorithms in a Heterogeneous Cluster of Personal Computers," In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000),* Cancun, Mexico, IEEE Computer Society Press, May 2000, pp.147-159.

[BWF+96] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks," In *Proceedings of Supercomputing '96*, November 1996.

[BWC+03] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, S. Spring, A. Su, and D. Zagorodnov, "Adaptive computing on the Grid using AppLeS," In *IEEE Transactions on Parallel and Distributed Systems (TPDS),* Volume 14, No. 4, pp.369-382, April 2003.

[CA96] Ü. V. Çatalyürek and C. Aykanat, "Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication," In *Parallel Algorithms for Irregularly Structured Problems, Irregular '96,* Volume 1117, Lecture Notes in Computer Science, pp.75-86. Springer Verlag 1996.

[CBL+02] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, and J. Ramanujam, "Memory-Constrained Communication Minimization for a Class of Array Computations," In *Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC '02),* College Park, Maryland, July 2002.

[CC99] A. Clematis and A. Corana, "Modeling Performance of Heterogeneous Parallel Computing Systems," In *Parallel Computing,* Volume 25, No. 9, pp.1131-1145, 1999.

[CD96] H. Casanova and J. Dongarra, “Netsolve: A Network Server for Solving Computational Science Problems,” In *Proceedings of the Supercomputing’96*, IEEE Computer Society Press, Los Alamitos, California, 1996.

[CDD+96] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, “ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance,” In *Computer Physics Communication*, Volume 97, pp.1-15, 1996.

[CDO+96] J. Choi, J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, “The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines,” In *Scientific Programming*, Volume 5, No. 3, pp.173–184, Fall 1996, ISSN 1058-9244.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick, “Parallel Programming in Split-C,” In *Proceedings of Supercomputing '93*, Portland, Oregon, November 1993, pp.262-273.

[CFM+01] F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg, “HiHCoHP: Toward a Realistic Communication Model for Hierarchical HyperClusters of Heterogeneous Processors.” In *Proceedings of the15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, 23-27 April 2001, San Francisco, California, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2001.

[CKP+93] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, “LogP: Towards a Realistic Model of Parallel Computation,” In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[CL94] Jr. P. Ciarlet and F. Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Technical Report 94-37*, Computer Science Department, UCLA, Los Angeles, CA, 9914.

[CLZ97] M. Cierniak, W. Li, and M. J. Zaki, “Compile-time scheduling algorithms for heterogeneous network of workstations,” In *Computer Journal*, special issue on Automatic Loop Parallelization, Volume 40, No. 6, December 1997.

[CP96] M. Coli and P. Palazzari, “Real time pipelined system design through simulated annealing,” In *Journal of Systems Architecture*, Volume 42, No. 6-7, , pp.465-475, December 1996.

[CQ93] P. Crandall and M. Quinn, “Block Data Decomposition for Data-Parallel Programming on a Heterogeneous Workstation Network,” In *Proceedings of the Second International Symposium on High Performance Distributed Computing*

(HPDC '93), Spokane, WA, USA, IEEE Computer Society, ISBN 0-8186-3900-4, pp.42-49, July 20-23, 1993.

[CQ95] P. Crandall and M. Quinn, “Problem Decomposition for Non-Uniformity and Processor Heterogeneity,” In *Journal of the Brazilian Computer Society*, Volume 2, No. 1, pp.13-23, July 1995.

[DBP94] V. Donaldson, F. Berman, and R. Paturi, “Program Speedup in a Heterogeneous Computing Network,” In *Journal of Parallel and Distributed Computing*, Volume 21, No. 3, pp.316-322, June 1994.

[DCD+90] J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, “A set of level-3 basic linear algebra subprograms,” In *ACM Transactions on Mathematical Software*, Volume 16, No. 1, pp.1-17, March 1990.

[Dim01] R. Dimitrov. Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations. *Ph.D. dissertation*, Department of Computer Science, Mississippi State University, May 2001.

[DL04] J. Dongarra and A. Lastovetsky, “An Overview of Heterogeneous High Performance and Grid Computing,” In Engineering the Grid, B. D. Martino, J. Dongarra, A. Hoisie, L. Yang, and H. Zima, editors, Nova Science Publishers, Inc, 2004.

[DVW94] J. Dongarra, R. A. van de Geijn, and D. W. Walker, “Scalability Issues Affecting the Design of a Dense Linear Algebra Library,” In *Journal of Parallel and Distributed Computing*, Volume 22, No. 3, pp.523-537, September 1994.

[DW03a] M. Drozdowski and P. Wolniewicz, “Divisible Load Scheduling in Systems with Limited Memory,” In *Cluster Computing*, Volume 6, No. 1, pp.19-29, January 2003.

[DW03b] M. Drozdowski and P. Wolniewicz, “Out-of-Core Divisible Load Processing,” In *IEEE Transactions on Parallel and Distributed Systems*, Volume 14, No. 10, pp.1048-1056, October 2003.

[FB96] S. M. Figueira and F. Berman, “Modeling the effects of contention on the performance of heterogeneous applications,” In *Proceedings of the High Performance Distributed Computing Conference (HPDC'96)*, Syracuse, New York, August 06 - 09, 1996, pp.392-402.

[FBD01] G. E. Fagg, A.Bukovsky, and J. Dongarra, “HARNESS and Fault Tolerant MPI,” In *Parallel Computing*, Volume 27, No. 11, pp.1479-1495, October 2001.

[Fer89] D. Fernandez-Baca, ‘‘Allocating modules to processors in a distributed system,’’ In *IEEE Transactions on Software Engineering*, Volume SE-15, No. 11, pp.1427-1436, November 1989.

[FGA+98] R. F. Freund, M. Gherrity, S. Ambrosius, M. Camp-bell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, "Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet," In *Proceedings of seventh IEEE Heterogeneous Computing Workshop (HCW '98)*, March 1998, pp.184-199.

[FGK+97] I. Foster, J. Geisler, C. Kesselman, and S. Tuecke, "Managing Multiple Communication Methods in High-Performance Networked Computing Systems," In *Journal on Parallel and Distributed Computing*, Volume 40, No.1, pp.35-48, January 1997.

[FK97] I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," In *International Journal of Supercomputer Applications*, Volume 11, No. 2, pp.115-128, Summer 1997.

[FK98] I. Foster, and C. Kesselman. ***The Grid***: *Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann Publishers, San Francisco, CA, 1998.

[FM82] C. M. Fiduccia and R. M. Mattheyses, "A linear time heuristic for improving network partitions," In Proceedings of 19th IEEE Design Automation Conference, pp.175-181, 1982.

[FW78] S.Fortune, and J.Wyllie, "Parallelism in Random Access Machines," In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pp.114-118, 1978.

[GAB+05] J. P.–Grbovi´c, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. Dongarra, "Performance Analysis of MPI Collective Operations," *submitted to Fourth International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS'05)*, Denver, Colorado, USA, April 4-8, 2005.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. ***PVM***: *Parallel Virtual Machine, Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press: Cambridge, MA, 1994.

[GL81] A. George and J. W.-H. Liu. Computer Solution of Large Sparse Positive Definite Systems. Prentice Hall, Englewood Cliffs, NJ, 1981.

[GL94] S. Gaissaryan, and A. Lastovetsky, "An ANSI C superset for vector and superscalar computers and its retargetable compiler," In *The Journal of C Language Translation*, Volume 5, No. 3, pp.183-198, March 1994.

[GLD+96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” In *Parallel Computing*, Volume 22, No. 6, pp.789-828, September 1996.

[GS94] T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. *Technical report*, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

[GW97] A. S. Grimshaw and W. A. Wulf, “The Legion Vision of a Worldwide Virtual Computer,” In *Communications of the ACM*, Volume 40, No. 1, pp.39-45, January 1997.

[HK00] B. Hendrickson and T. G. Kolda, “Partitioning sparse rectangular and structurally nonsymmetric matrices for parallel computation,” In *SIAM Journal on Scientific Computing*, Volume 21, No. 6, pp.2048-2072, May 2000.

[HKJ+99] D. A. Hensgen, T. Kidd, D. St. John, M. C. Schnaidt, H. J. Siegel, T. D. Braun, M. Maheswaran, S. Ali, J. Kim, C. Irvine, T. Levin, R. F. Freund, M. Kussow, M. G. A. Duman, P. Carff, S. Kidd, V. Prasanna, P. Bhat, and A. Alhusaini, “An overview of MSHN: The Management System for Heterogeneous Networks,” In *8th Heterogeneous Computing Workshop (HCW '99)*, April 1999, pp.184-198.

[HL93a] B. Hendrickson and R. Leland, “A multilevel algorithm for partitioning graphs,” *Technical Report SAND93-1301*, Sandia National Laboratories, 1993.

[HL93b] B. Hendrickson and R. Leland, “An improved spectral load balancing method,” In *Proceedings of 6th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, March 1993, pp.953-961.

[HL94] B. Hendrickson and R. Leland, “The Chaco user's guide, version 2.0,” *Technical Report SAND94-2692*, Sandia National Laboratories, 1994.

[HPF94] High Performance Fortran Forum, High Performance Fortran Language Specification, version 1.1, 1994.

[HPF97] High Performance Fortran Forum, High Performance Fortran Language Specification, version 2.0, 1997.

[HQ91] P. J. Hatcher and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. MIT Press, Cambridge, MA, 1991.

[HW94] B. Hendrickson, and D. Womble, “The Torus-Wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers,” In *SIAM Journal on Scientific and Statistical Computing*, Volume 15, No. 5, pp.1201-1226, September 1994.

[IK77] O. H. Ibarra and C. E. Kim, “Heuristic algorithms for scheduling independent tasks on non-identical processors," In *Journal of the ACM*, Volume 24, No. 2, pp.280-289, April 1977.

[IO98] M. Iverson and F. Ozguner, "Dynamic, Competitive Scheduling of Multiple DAGs in a Distributed Heterogeneous Environment," In *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, Orlando, FL, IEEE Computer Society Press, pp.70-78, March 1998.

[JGN99] W. Johnston, D. Gannon, and B. Nitzberg, "Grids as production computing environments: The engineering aspects of nasa's information power grid," In *Proceedings of Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.

[KAK+97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design," In *Proceedings of 34th ACM/IEEE Design Automation Conference*, Anaheim, California, USA, pp.526-529, June 9-13, 1997, ACM Press, 1997.

[KBG00] T. Kielmann, H. E. Bal, and S. Gorlatch, "Bandwidth-Efficient Collective Communication for Clustered Wide Area Systems," In *International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, May 1-5, 2000, pp.492-499.

[KBV00] T. Kielmann, H. Bal, and K. Verstoep, "Fast measurement of LogP parameters for message passing platforms," In Proceedings of the fourth Workshop on runtime systems for Parallel Programming (RTSPP'00), Lecture Notes in Computer Science, Volume 1800, Springer Verlag, Cancun, Mexico, 2000, pp.1176-1183.

[KDB02] S. Kumar, S. K. Das, and R. Biswas, "Graph Partitioning for Parallel Applications in Heterogeneous Grid Environments," In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, 15-19 April 2002, Fort Lauderdale, California, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2002.

[KGG+94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, Addison-Wesley, 597 pages, 1994.

[KGV83] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," In *Science*, Volume 200, No. 4598, pp.671-680, May 1983.

[KI04] Y. Kishimoto and S. Ichikawa, "An Execution-Time Estimation Model for Heterogeneous Clusters," In *13th Heterogeneous Computing Workshop (HCW 2004), in Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, IEEE Computer Society (2004).

[KK93] L.V. Kalé, and S. Krishnan, “CHARM++: A Portable Concurrent Object Oriented System Based on C++,” In *Proceedings of OOPSLA'93*, pp.91-108, ACM Press, 1993.

[KK95] G. Karypis and V. Kumar, “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs,” *Technical Report TR 95-035*, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[KK97] G. Karypis and V. Kumar, “A coarse-grain parallel formulation of a multilevel k-way graph partitioning algorithm,” In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[KK98a] G. Karypis and V. Kumar, “Multilevel Algorithms for Multi-constraint Graph Partitioning”, *Technical Report TR 98-019*, Department of Computer Science and Engineering, University of Minnesota, 1998.

[KK98b] G. Karypis and V. Kumar, “Multilevel k-way Hypergraph Partitioning,” In *Proceedings of 36th ACM/IEEE Design Automation Conference*, New Orleans, LA, USA, ACM Press, June 21-25, 1999, pp.343-348.

[KL70] B. W. Kernighan and S. Lin. An Efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*. 1970.

[KL01] A. Kalinov and A. Lastovetsky, “Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers,” In *Journal of Parallel and Distributed Computing*, Volume 61, No. 4, pp.520-535, April 2001.

[KRW96] M. Kaddoura, S. Ranka, and A. Wang, “Array decompositions for nonuniform computational environments,” In *Journal of Parallel and Distributed Computing*, Volume 36, No. 2, pp.91-105, August 1996.

[KS01] J. Ke and E. Speight, “Tern: Migrating threads in an MPI runtime environment,” *Technical Report SCL-TR-2001-1016*, Cornell, November 2001.

[HLD97] B. Hendrickson, R. Leland, and R. V. Driessche, “Skewed graph partitioning,” *In Proceedings of 8th SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Minneapolis, Minnesota, March 1997.

[LAK+00] A. Lastovetsky, D. Arapov, A. Kalinov, and I. Ledovskih, “A Parallel Language and Its Programming System for Heterogeneous Networks,” In *Concurrency: Practice and Experience*, Volume 12, No. 13, pp.1317-1343, November 2000.

[Las02] A. Lastovetsky, “Adaptive Parallel Computing on Heterogeneous Networks with mpC,” In *Parallel Computing*, Volume 28, No. 10, pp.1369-1407, October 2002.

[Las03] A. Lastovetsky. *Parallel Computing on Heterogeneous Networks*. John Wiley & Sons, 423 pages, 2003.

[LB96] B. Lowekamp, and A. Beguelin, "ECO: Efficient Collective Operations for Communication on Heterogeneous Networks," In *International Parallel Processing Symposium*, Honolulu, HI, 1996, pp.399-405.

[LBK02] O. Lawlor, M. Bhandarkar, and L. V. Kale, "Adaptive MPI," *Technical Report TR 02-05*, University of Illinois, 2002.

[LLM88] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-A Hunter of Idle Workstations, In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 1988, pp.104-111.

[LNL+00] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou, "MPI-FT: Portable Fault Tolerance Scheme for MPI," In *Parallel Processing Letters*, Volume 10, No. 4, pp. 371-382, December 2000.

[LT04] A. Lastovetsky and J. Twamley, "Towards a Realistic Performance Model for Networks of Heterogeneous Computers," In *International Symposium on High Performance Computational Science and Engineering (HPSCE'04)*, Toulouse, France, August 22-27, 2004.

[LVK00] X. Li, B. Veeravalli, and C.C. Ko, "Divisible Load Scheduling on Single-level Tree Networks with Finite-size Buffers," In *IEEE Transactions on Aerospace and Electronic Systems*, Volume 36, No. 4, pp.1298-1308, October 2000.

[MAS+99a] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, and R.F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," In *Eight Heterogeneous Computing Workshop*, pages 30-44. IEEE Computer Society Press, 1999.

[MAS+99b] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic Mapping of a Class of Independent Tasks onto Heterogeneous Computing Systems," In *Journal of Parallel and Distributed Computing*, Volume 59, No. 2, pp.107-131, November 1999.

[MF98] C. A. Moritz, and M. I. Frank, "LoGPC: Modeling network contention in message-passing programs," In *Proceedings of Sigmetrics*, 1998.

[Mor98] P. Morin, "Coarse grained parallel computing on heterogeneous systems," In *Proceedings of the 13th Annual ACM Symposium on Applied Computing (SAC'98)*, 1998, pp.629-634.

[MS98a] M. Maheswaran, and H. J. Siegel, "A dynamic matching and scheduling algorithm for heterogeneous computing systems," In *Proceedings of the Seventh*

Heterogeneous Computing Workshop (HCW 1998), Orlando, FL, IEEE Computer Society Press, March 1998, pp.57-69.

[MS98b] J. Marinho and J. G. Silva, “WMPI - Message Passing Interface for Win32 Clusters,” In *Proceedings of 5th European PVM/MPI User's Group Meeting*, September 1998, pp.113-129.

[MTV91] G. L. Miller , S-H. Teng, and S. A. Vavasis, “A unified geometric approach to graph separators, In *Proceedings of the 32nd annual symposium on Foundations of computer science*, San Juan, Puerto Rico, September 1991, pp.538-547.

[NT93] M. G. Norman and P. Thanisch, “Models of Machines and Computation for Mapping in Multicomputers,” In *ACM Computing Surveys*, Volume 25, No. 3, pp.263-302, September 1993.

[OMG98] Object Management Group. The Common Object Request Broker: Architecture and Specification. Revision 2.3. 1998.

[PCA+96] A. Pinar, Ü. V. Çatalyürek, C. Aykanat, and M. Pinar, “Decomposing linear programs for parallel solution,” In *Applied Parallel Computing in Computations in Physics, Chemistry, and Engineering Science, PARA '95*, Lecture Notes in Computer Science, Volume 1041, pp.473-482. Springer Verlag 1996.

[PD99] A. Petitet and J. Dongarra, "Algorithmic Redistribution Methods for Block-Cyclic Decompositions," In *IEEE Transactions on Parallel and Distributed Systems*, Volume 10, No. 12, pp.1201-1216, December 1999.

[Pel94] F. Pellegrini, “Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs,” In *Proceedings of SHPCC'94*, Knoxville, 1994, pp.486-493.

[PG01] E. Post and H. A. Goosen, “Evaluating the Parallel Performance of a Heterogeneous System,” In *HPC Asia 2001, 5th International Conference and Exhibition on High-Performance Computing* in the Asia-Pacific Region, Gold Coast, Australia, September 2001.

[PS81] Y. Perl and S. R. Schach, “Max-min tree partitioning,” In *Journal of the ACM*, Volume 28, No. 1, pp.5-15, January 1981.

[PSL90] A. Pothen, H. D. Simon, and K-P. Liou, “Partitioning sparse matrices with eigenvectors of graphs,” In *SIAM Journal on Matrix Analysis and Applications*, Volume 11, No.3, pp.430-452, July 1990.

[Rag93] P. Raghavan. Line and Plane separators. *Technical Report UIUCDCS-R-93-1794*, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

[RD01] U. Rencuzogullari, and S. Dwarkadas, “Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations,” In *Eighth ACM PPOPP*, pp.72-81, June 2001.

[RN95] S. J. Russell and P. Norwig. Artificial Intelligence: A Modern Approach, Prentice Hall, Englewood Cliffs, NJ, 1995.

[RS99] R. Dimitrov and A. Skjellum, “An efficient MPI implementation for Virtual Interface Architecture – enabled cluster computing,” In *Proceedings of the 3rd MPI developer’s and user’s conference*, Atlanta, Georgia, March 1999, pp.15-24.

[SDA96] Howard J. Siegel, Henry G. Dietz, and John K. Antonio, “Software Support for Heterogeneous Computing,” In *ACM Computing Surveys*, Volume 28, No. 1, pp. 237-239, March 1996.

[SKK00] Kirk Schloegel, George Karypis, and Vipin Kumar, “A unified algorithm for load-balancing adaptive scientific simulations,” In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (CDROM), p.59-es, November 04-10, 2000, Dallas, Texas, United States.

[SKK99a] K. Schloegel, G. Karypis, and V. Kumar, “A new algorithm for multi-objective graph partitioning,” In *Proceedings of EuroPar '99*, Lecture Notes in Computer Science, Springer-Verlag, 1999.

[SKK99b] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. *Technical Report TR 99-031*, Department of Computer Science and Engineering, University of Minnesota, 1999.

[SOJ+96] M. Snir, S. Otto, S. Huss-Lederman, David Walker, and J. Dongarra. ***MPI**: The Complete Reference*. The MIT Press, 1996.

[Sto77a] H. S. Stone, “Multiprocessor scheduling with the aid of network flow algorithms,” In *IEEE Transactions on Software Engineering*, Volume SE-3, No. 1, pp.85-93, January 1977.

[Sto77a] H. S. Stone, “Program assignment in three-processor systems and tricutset partitioning of graphs,” *Technical Report ECE-CS-77-7*, Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, July 1977.

[Sto78] H. S. Stone, “Critical load factors in two-processor distributed systems,” In *IEEE Transactions on Software Engineering*, Volume SE-4, pp.254-258, May 1978.

[SP94] M. Srinivas and L. M. Patnaik, “Genetic algorithms: A survey,” In *IEEE Computer*, Volume 27, No. 6, pp.17-26, June 1994.

[Sun02] X.-H.Sun, "Scalability versus Execution Time in Scalable Systems", In *Journal of Parallel and Distributed Computing*, Volume 62, No. 2, pp.173-192, 2002.

[SW03] X.-H. Sun, and M. Wu, "Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling," In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003),* Nice, France, April, 2003.

[SY96] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," In *5th IEEE Heterogeneous Computing Workshop (HCW '96),* April 1996, pp.86-97.

[Tol99] S. Toledo, "A survey of out-of-core algorithms in numerical linear algebra," In *External Memory Algorithms and Visualization,* J. Abello and J. S. Vitter, Eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society Press, Providence, RI, 1999.

[TSA+97] M. Tan, H. J. Siegel, J. K. Antonio, and Y. A. Li, "Minimizing the Application Execution Time through Scheduling of Subtasks and Communication Traffic in a Heterogeneous Computing System", In *IEEE Transactions on Parallel and Distributed Systems,* Volume 8, No. 8, pp.857-871, August 1997.

[Val90] L. G. Valiant, "A Bridging Model for Parallel Computation," In *Communications of the Association for Computing Machinery,* Volume 33, No. 8, pp.103-111, 1990.

[VFD00a] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications," In *Proceedings of SC2000,* Dallas TX, November 2000.

[VFD04b] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Towards an Accurate Model for Collective Communications," In *International Journal of High Performance Computing Applications, Special Issue: Automatic Performance Tuning,* Volume 18, No. 1, pp.159-167, Spring 2004, ISSN 1094-3420.

[WC00] C. Walshaw and M. Cross, "Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm," In *SIAM Journal of Scientific Computing*, Volume 22, pp.63-80, 2000.

[Wil00] T. L. Williams. A General-Purpose Model for Heterogeneous Computation. Ph.D. dissertation, University of Central Florida, Orlando, December 2000.

[WLN+03] D. B. Weatherly, D. K. Lowenthal, M. Nakazawa, and F. Lowenthal, "Dyn-MPI: Supporting MPI on a Nondedicated Cluster of Workstations," In *Proceedings of the 15th IEEE/ACM Supercomputing 2003 (SC'03),* Phoenix, AZ, April 2003.

[WP00] T. L. Williams, and R. J. Parsons, “The heterogeneous bulk synchronous parallel model,” In *Parallel and Distributed Processing*, Lecture Notes in Computer Science, Volume 1800, Springer Verlag, Cancun, Mexico, May 2000, pp.102-108.

[WPD00] R. C. Whaley, A. Petitet, and J. Dongarra, “Automated empirical optimizations of software and the atlas project,” Technical report, Department of Computer Sciences, University of Tennessee, Knoxville, March 2000.

[WS97] R. Wolski and N. Spring, “Implementing a Performance Forecasting System for Metacomputing: The Network Weather Service,” In *Proceedings of the IEEE/ACM SC97 Conference*, San Jose, California, November 15 - 21, 1997.

[WS01] M.-Y. Wu, and W. Shu, “A High-Performance Mapping Algorithm for Heterogeneous Computing Systems,” In *Proceedings of the15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, 23-27 April 2001, San Francisco, California, CD-ROM/Abstracts Proceedings, IEEE Computer Society 2001.

[WS04] M. Wu, and X.-H. Sun, “Memory Conscious Task Partition and Scheduling in Grid Environments,” In *5th IEEE/ACM International Workshop on Grid Computing (in conjunction with SuperComputing 2004)*, Pittsburgh, pp.138-145, November 2004.

[WSR+97] L.Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, “Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach,” In *Journal of Parallel and Distributed Computing*, Volume 47, No. 1, pp.1-15, November 1997.

[WSS00] R. Wolski, N.T. Spring, and J. Spring, “Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid,” In Cluster Computing, Volume 3, No. 4, pp.293-301, 2000.

[XCZ02] L. Xiao, S. Chen, and X. Zhang, “Dynamic Cluster Resource Allocations for Jobs with Known and Unknown Memory Demands,” In *IEEE Transactions on Parallel and Distributed Systems*, Volume 13, No. 3, pp.223-240, March 2002.

[YWC+95] K. Yelick, C. Wen, S. Chakrabarti, E. Deprit, J. A. Jones, and A. Krishnamurthy, “Portable Parallel Irregular Applications,” In *Workshop on Parallel Symbolic Languages and Systems (PLPS’95)*, Beaune, France, October 2-4, 1995, Lecture Notes in Computer Science, 1068, pp.157-173, 1996.

[YZS96] Y. Yan, X. Zhang, and Y. Song, “An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW,” In *Journal of Parallel and Distributed Computing*, Volume 38, No. 1, pp.63-80, October 1996.

[ZY95] X. Zhang, and Y. Yan, “Modeling and Characterizing Parallel Computing Performance on Heterogeneous Networks of Workstations”, In *Proceedings of the seventh IEEE Symposium on Parallel and Distributed Processing (SPDP'95)*, October, 1995.

APPENDIX A

Generated Code from Compiling Performance Model Definition of Parallel Simulation of Evolution of Bodies

```
/* This file was generated by mpC compiler
   From file "Nbody.mpc"*/

static char* MPC_file_name="Nbody.mpc";

#include <mpC.h>
#include <topo.h>
#include <mpc_macro.h>
static MPC_Basic MPC_int={{kMPC_Basic,"const int ",3,sizeof(const int
),1,MPC_DTN},7};

/*"Nbody.mpc"*/
typedef double Triplet [3];
typedef struct {
  Triplet p;
  Triplet v;
  double m;
} Body ;

/* Net type Nbody declaration */
int MPC_NetType_Nbody_node(int pnum,const int *ppar,
       int ppower,int **pnodes,int **plinks);

int MPC_NetType_Nbody_link(int pnum1,int pnum2,
       const int *ppar,int ppower,int **pnodes,
       int **plinks);

int MPC_NetType_Nbody_parent(const int *ppar,
       int ppower,int **pnodes,int **plinks);

int MPC_NetType_Nbody_power(const int *ppar,int ppower,
       int **pnodes,int **plinks);

void MPC_NetType_Nbody_number2coord(int pnum,
       const int *ppar,int *pcoord,int ppower,
       int **pnodes,int **plinks);

int MPC_NetType_Nbody_coord2number(const int *pcoord,
       const int *ppar,int ppower,int **pnodes,
       int **plinks);

double MPC_NetType_Nbody_mapping(
       MPC_Topo_graph **root,const int *ppar,int ppower,
       int **pnodes,int **plinks);

int MPC_NetType_Nbody_node(int pnum,const int *ppar,
       int ppower,int **pnodes,int **plinks) {
  static MPC_Pointer MPC_pointer_7={{kMPC_Pointer,
          "void *",7,sizeof(void* ),1,MPC_DTN},1,NoMPC_Type};
  int coordinate[1];
  MPC_NetType_Nbody_number2coord(pnum,ppar,coordinate,
           ppower,pnodes,plinks);
  if ( * coordinate >= 0)
  {
     return (int )(MPC_FIXPOINT_SCALE * (( * ((ppar + 2) +  *
coordinate) /  * (ppar + 1)) * ( * ((ppar + 2) +  * coordinate) /  *
(ppar + 1))));
```

```
  }
  return 0;
}

int MPC_NetType_Nbody_link(int pnum1,int pnum2,const int *ppar,
       int ppower,int **pnodes,int **plinks) {
  int coordinate[1];
  int c1[1],n1;
  int c2[1],n2;
  void *calloc();
  if (ppower == MPC_INIT_POWER)
  {
     ppower = MPC_NetType_Nbody_power(ppar,ppower,pnodes,plinks);
  }

  if ( ! ( * plinks))
  {
     * plinks = (int *)calloc(ppower * ppower,sizeof(int ));
     for (coordinate[0] = 0; coordinate[0] < *ppar; coordinate[0] ++) {
        if ( * coordinate > 0) {
            c1[0] =  * coordinate;
            c2[0] = 0;
            n1 = MPC_NetType_Nbody_coord2number(c1,ppar,ppower,
                    pnodes,plinks);
            n2 = MPC_NetType_Nbody_coord2number(c2,ppar,ppower,
                    pnodes,plinks);
            ( * plinks)[n1 * ppower + n2] = ( * ((ppar + 2) +
                                        * coordinate) * sizeof(Body ));
        }
      }
    }
  }
  return ( * plinks)[pnum1 * ppower + pnum2];
}

int MPC_NetType_Nbody_parent(const int *ppar,int ppower,
       int **pnodes,int **plinks) {
  int coordinate[1];
  coordinate[0] = 0;
  return MPC_NetType_Nbody_coord2number(coordinate,ppar,
            ppower,pnodes,plinks);
}

int MPC_NetType_Nbody_power(const int *ppar,int ppower,
       int **pnodes,int **plinks) {
  return  * ppar;
}

void MPC_NetType_Nbody_number2coord(int pnum,const int *ppar,
       int *pcoord,int ppower,int **pnodes,int **plinks) {
  * pcoord = pnum;
}

int MPC_NetType_Nbody_coord2number(const int *pcoord,
       const int *ppar,int ppower,int **pnodes,int **plinks) {
  return  * pcoord;
}
```

```
double MPC_NetType_Nbody_mapping(MPC_Topo_graph **root,
       const int *ppar,int ppower,int **pnodes,int **plinks) {
  const int p= * ppar,k= * (ppar + 1),*n=(ppar + 2);
  double MPC_estimation1=0.0;
  {
    int i;
    {
      double *MPC_estimation2,MPC_Max_estimation2=0.0;
      int MPC_actions2=0,*MPC_sorted_actions;
      char **MPC_involved_VPs2;
      MPC_estimation2=(double *)calloc
                      (ppower*(ppower+1),sizeof(double));
      MPC_involved_VPs2=(char **)calloc(ppower*(ppower+1),
                       sizeof(char *));
      for (i = 0;i < p;i ++ ) {
          MPC_involved_VPs2[MPC_actions2]=(char *)
             calloc(ppower,sizeof(char));
          {
             int MPC_coord[1],MPC_rank;
             MPC_coord[0] = i;
             MPC_rank=MPC_NetType_Nbody_coord2number(MPC_coord,ppar,
                         ppower,pnodes,plinks);
             MPC_estimation2[MPC_actions2]+=MPC_Part_comp_est
                         ((double)(100),root[MPC_rank]);
             MPC_involved_VPs2[MPC_actions2][MPC_rank]=1;
          }
          MPC_actions2++;
      }
      MPC_estimation1+=MPC_Par_estimation(ppower,MPC_actions2,
                       MPC_estimation2,MPC_involved_VPs2,NULL);
    }
    {
      double *MPC_estimation2,MPC_Max_estimation2=0.0;
      int MPC_actions2=0,*MPC_sorted_actions;
      char **MPC_involved_VPs2;
      MPC_estimation2=(double *)calloc
               (ppower*(ppower+1),sizeof(double));
      MPC_involved_VPs2=(char **)calloc(ppower*(ppower+1),
               sizeof(char *));
      for (i = 0;i < p;i ++ ) {
         MPC_involved_VPs2[MPC_actions2]=(char *)calloc
            (ppower,sizeof(char));
         {
            int MPC_coord[1],MPC_rank_from,MPC_rank_to;
            MPC_coord[0] = i;
            MPC_rank_from=MPC_NetType_Nbody_coord2number(MPC_coord,
                    ppar,ppower,pnodes,plinks);
            MPC_coord[0] = 0;
            MPC_rank_to=MPC_NetType_Nbody_coord2number(MPC_coord,
                    ppar,ppower,pnodes,plinks);
          MPC_estimation2[MPC_actions2]+=MPC_Part_comm_est(
               (double)(100),root[MPC_rank_from],root[MPC_rank_to]);
         }
         MPC_actions2++;
      }
      MPC_estimation1+=MPC_Par_estimation(ppower,
         MPC_actions2,MPC_estimation2,MPC_involved_VPs2,NULL);
    }
  }
  return MPC_estimation1;
}

MPC_NetType MPC_NetType_Nbody={1,
                              MPC_NetType_Nbody_node,
                              MPC_NetType_Nbody_link,
                              MPC_NetType_Nbody_parent,
                              MPC_NetType_Nbody_power,
                              MPC_NetType_Nbody_number2coord,
                              MPC_NetType_Nbody_coord2number,
                              MPC_NetType_Nbody_mapping};

#define HMPI_Model_Nbody MPC_NetType_Nbody
```

APPENDIX B

HMPI Programmers' Reference and Installation Manual

HMPI
A Message Passing Library for Heterogeneous Networks of Computers
Version 1.0

Ravi Reddy, Alexey Lastovetsky

Department of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

E-mail: Manumachu.Reddy@ucd.ie, Alexey.Lastovetsky@ucd.ie

March 18, 2005

CONTENTS

1 Introduction

The tools designed for programming high-performance computations on HNOCs must provide mechanisms to automate the tedious and error-prone tasks:

- Parameter determination characterizing the computational requirements of the parallel application and the capabilities of the machines,
- Data partitioning,
- Matching and Scheduling, and
- Task execution.

Ideally a tool must supply mechanisms to the programmer so that he or she can provide information to it that can assist in finding the most efficient implementation on HNOCs. Combining the system's detailed analysis with the programmer's high-level knowledge of the application is essential in finding more efficient mappings than either one alone is capable of achieving. The performance models used by the tools must take into account all the essential features underlying applications run on HNOCs, mainly, the speeds of the processors, the effects of paging and the speed and the bandwidth of the communication links between the processors. The model of the executing network of computers must take into consideration the essential set of machine characteristics such as computing bandwidth, communication latency, communication overhead, communication bandwidth, network contention effects and the memory hierarchy. Such a model must have enough parameters for it to be effective and accurate.

HMPI is such a tool, which is an extension of MPI for programming high-performance computations on heterogeneous networks of computers. The main idea of HMPI is to automate the process of selection of a group of processes, which would execute the heterogeneous algorithm faster than any other group. HMPI provides features that allow the user to carefully design their parallel applications that can run efficiently on HNOCs.

The rest of the manual is organized as follows. Section 2 describes HMPI. Section 3 presents the HMPI API, which are extensions to MPI. Section 4 presents the library of data partitioning functions. Section 5 provides the HMPI command-line user's interface. This is followed by installation instructions for HMPI for UNIX in section 6.

2 What is HMPI

Heterogeneous MPI (**HMPI**) is an extension of MPI for programming high-performance computations on heterogeneous networks of computers. It allows the application programmer to describe the performance model of the implemented algorithm in a generic form. This model allows for all the main features of the underlying parallel algorithm, which have an impact on its execution performance, such as the total number of parallel processes, the total volume of computations to be performed by each process, the total volume of data to be transferred between each pair of the processes, and how exactly the processes interact during the execution of the algorithm. Given the description of the performance model, HMPI tries to create a group of processes that executes the algorithm faster than any other group of processes.

HMPI provides all the features to the user to write portable and efficient parallel applications on HNOCs. These features automate all the essential steps involved in application development on HNOCs:

1). Determination of the characterization parameters relevant to the computational requirements of the applications and the machine capabilities of the heterogeneous system. The machine capabilities are determined before the application execution and supplied to the model of executing network of computers. The model of the executing network of computers is implementation-dependent. We use a static structure automatically obtained by HMPI environment and saved in the form of an ASCII file. However, the parameters of the model can be updated at runtime taking into account the changing network loads.

2). Decomposition of the whole problem into a set of sub-problems that can be solved in parallel by interacting processes. This step of heterogeneous decomposition is parameterized by the number and speeds of processors and the speeds and bandwidths of the communication links between them. The Heterogeneous Data Partitioning Interface (HDPI) is developed to automate this step of heterogeneous decomposition. HDPI provides API that allows the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition the mathematical objects used in their parallel applications.

3). Selection of the optimal set of processes running on different computers of the heterogeneous network by taking into account the speeds of the processors, and the speeds and the bandwidths of the communications links between them. During the creation of this set of optimal processes, HMPI runtime system solves the problem of selection of the optimal set of processes running on different computers of the heterogeneous network using an advanced mapping algorithm. The mapping algorithm is based on the performance model of the parallel algorithm in the form of the set of functions generated by the compiler from the description of the performance model, and the performance model of the executing network of computers, which reflects the state of this network just before the execution of the parallel algorithm.

4). Application program execution on the HNOCs. The command line user interface of HMPI developed consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of the HMPI application programs on the virtual parallel machine. The notion of virtual parallel machine enables a collection of heterogeneous computers to be used as single large parallel computer.

3 HMPI's Library Interface

In this section, we describe the interfaces to the routines provided by HMPI as extensions to MPI and the interfaces to the routines in the heterogeneous data partitioning interface (HPDI).

3.1 HMPI runtime initialization and finalization

`HMPI_Init`

Initializes HMPI runtime system

Synopsis:

```
int
HMPI_Init
(
    int argc,
    char** argv
)
```

Parameters:

`argc` --- Number of arguments supplied to **`main`**
`argv` --- Values of arguments supplied to **`main`**

Description: All processes must call this routine to initialize HMPI runtime system. This routine must be called before any other HMPI routine. It must be called at most once; subsequent calls are erroneous.

Usage:

```
int main(int argc, char** argv)
{
   int rc =  HMPI_Init(
                 argc,
                 argv
   );

   if (rc != HMPI_SUCCESS)
   {
      //Error has occurred
   }
}
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

HMPI_Finalize

Finalizes HMPI runtime system

Synopsis:

```
int
HMPI_Finalize
(
    int exitcode
)
```

Parameters:

`exitcode` --- code to be returned to the command shell

Description: This routine cleans up all HMPI state. All processes must call this routine at the end of processing tasks. Once this routine is called, no HMPI routine (even **`HMPI_Init`**) may be called.

Usage:

```
int main(int argc, char** argv)
{
    int rc =  HMPI_Init(
                  argc,
                  argv
    );

    if (rc != HMPI_SUCCESS)
    {
       //Error has occurred
    }

    rc =  HMPI_Finalize(0);

    if (rc != HMPI_SUCCESS)
    {
       //Error has occurred
    }
}
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

3.2 HMPI Group Management Functions

`HMPI_Group_rank`

Returns rank of the calling process

Synopsis:

```
int
HMPI_Group_rank
(
    const HMPI_Group* gid
)
```

Parameters:

`gid` --- handle to the HMPI group of processes

Description: This routine returns the rank of the process calling it. Only processes that are members of the group represented by the handle **`gid`** can call this routine.

Usage:

```
    // HMPI_HOST_GROUP is a predefined group handle
    // containing the host process.
    HMPI_Group* gid = HMPI_HOST_GROUP;

    if (HMPI_Is_member(gid))
    {
       int rank =  HMPI_Group_rank(
                       gid
       );
    }
```

Return values: Error code `HMPI_UNDEFINED` is returned if the process is not the member of the group represented by the handle `gid`.

HMPI_Rank

Returns rank of the process with the coordinates specified

Synopsis:

```
   int
   HMPI_Rank
   (
       const HMPI_Group* gid,
       const int* coordinates
   )
```

Parameters:

`gid`	--- handle to the HMPI group of processes
`coordinates`	--- coordinates representing a process in the group represented by the handle `gid`

Description: This routine returns the rank of the process in the group represented by the handle `gid` and the coordinates of the process being `coordinates`. Only processes that are members of the group represented by the handle `gid` can call this routine.

Usage:

```
    // HMPI target program
    HMPI_Group gid;
    int coordinates = 3;
    if (HMPI_Is_member(&gid))
```

```
   {
      int rank = HMPI_Rank(
                    &gid,
                    &coordinates
      );
   }
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group represented by the handle **`gid`**.

HMPI_Group_coordof

Returns the coordinates of the process

Synopsis:

```
   int
   HMPI_Group_coordof
   (
      const HMPI_Group* gid,
      int* numc,
      int** coordinates
   )
```

Parameters:

`gid` --- Handle to the HMPI group of processes. This is an input parameter.
`numc` --- Output parameter giving the number of coordinates representing the calling process in the group represented by the handle **`gid`**.
`coordinates` --- The values of the coordinates of the calling process in the group represented by the handle **`gid`**.

Description: If the process calling this routine is a member of the group given by the handle **`gid`**, then its coordinates are returned in **`coordinates`**, the initial element of which points to an integer array containing the coordinates with size **`*numc`**. Only processes that are members of the group represented by the handle **`gid`** can call this routine.

Usage:

```
   HMPI_Group gid;

   if (HMPI_Is_member(&gid))
   {
      int numc;
      int** coordinates = (int**)malloc(
                                sizeof(int*)
      );

      int rc =  HMPI_Group_coordof(
                    &gid,
                    &numc,
                    coordinates
      );

      if (rc != HMPI_SUCCESS)
      {
         //Failure
      }

      free(coordinates[0]);
      free(coordinates);
   }
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group represented by the handle **`gid`**. **`HMPI_SUCCESS`** is returned on success.

HMPI_Coordof

Returns the coordinates of the process with a specified rank.

Synopsis:

```
   int
   HMPI_Coordof
   (
       const HMPI_Group* gid,
       int rank,
       int* numc,
       int** coordinates
   )
```

Parameters:

`gid` --- Handle to the HMPI group of processes. This is an input parameter.
`rank` --- The rank of the process whose coordinates are returned. This is an input parameter.
`numc` --- Output parameter giving the number of coordinates of the process whose rank is **`rank`** in the group represented by the handle **`gid`**.
`coordinates` --- The values of the coordinates of the process whose rank is **`rank`** in the group represented by the handle **`gid`**.

Description: The coordinates of the process whose rank is **rank** in the group represented by the handle **gid** are returned in **coordinates**, the initial element of which points to an integer array containing the coordinates with size ***numc**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(gid))
{
   int rank = 0;
   int numc;
   int** coordinates = (int**)malloc(
                           sizeof(int*)
   );

   int rc = HMPI_Coordof(
                &gid,
                rank,
                &numc,
                coordinates
   );

   if (rc != HMPI_SUCCESS)
   {
      //Failure
   }

   free(coordinates[0]);
   free(coordinates);
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**. **HMPI_SUCCESS** is returned on success.

HMPI_Group_topo_size

Returns the number of coordinates that can specify a process in a group

Synopsis:

```
int
HMPI_Group_topo_size
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HMPI group of processes

Description: This routine returns the number of coordinates used to specify a process, which is a member of the group represented by the handle **gid**. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(gid))
{
   int numc =  HMPI_Group_topo_size(
                   &gid
   );
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**.

HMPI_Group_topology

Returns the number of processes in the group in each dimension of the topology of the group.

Synopsis:

```
int
HMPI_Group_topology
(
    const HMPI_Group* gid,
    int* numc,
    int** coordinates
)
```

Parameters:

gid --- handle to the HMPI group of processes.
numc --- Output parameter giving the number of dimensions of the topology specifying the arrangement of the processes, which are members of the group represented by the handle **gid**.
coordinates --- Output parameter giving the number of processes in each dimension of the topology specifying the arrangement of

the processes, which are members of the group represented by the handle **gid**.

Description: This routine returns the number of dimensions of the topology and the number of processes in each dimension of the topology representing the arrangement of the processes, which are members of the group represented by the handle **gid**. The number of processes in each dimension are returned in **coordinates**, the initial element of which points to an integer array with number ***numc** of elements containing the number of dimensions. Only processes that are members of the group represented by the handle **gid** can call this routine.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(&gid))
{
   int numc;
   int** coordinates = (int**)malloc(
                           sizeof(int*)
   );
   int rc =  HMPI_Group_topology(
                 &gid,
                 &numc,
                 coordinates
   );

   if (rc != HMPI_SUCCESS)
   {
      //Failure
   }

   free(coordinates[0]);
   free(coordinates);
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group represented by the handle **gid**. **HMPI_SUCCESS** is returned on success.

HMPI_Group_parent

Returns the rank of the parent of a group

Synopsis:

```
int
```

```
HMPI_Group_parent
(
    const HMPI_Group* gid
)
```

Parameters:

`gid` --- handle to the HMPI group of processes.

Description: This routine returns the rank of the parent of the group represented by the handle **`gid`**. Only processes that are members of the group represented by the handle **`gid`** can call this routine.

Usage:

```
HMPI_Group* gid;
int rank;
if (HMPI_Is_member(gid))
{
   rank =  HMPI_Group_parent(gid);
}
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group represented by the handle **`gid`**.

HMPI_Group_size

Returns the number of processes in the group

Synopsis:

```
int
HMPI_Group_size
(
    const HMPI_Group* gid
)
```

Parameters:

`gid` --- handle to the HMPI group of processes

Description: This routine returns the number of processes in the group represented by the handle **`gid`**. Only processes that are members of the group represented by the handle **`gid`** can call this routine.

Usage:

```
HMPI_Group* gid;
int size;

if (HMPI_Is_member(gid))
{
   size = HMPI_Group_size(gid);
}
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group represented by the handle **`gid`**.

`HMPI_Is_host`

Is the calling process the host?

Synopsis:

```
unsigned char
HMPI_Is_host()
```

Description: This routine returns **`true`** if the process calling this function is the host process otherwise **`false`**. Any process can call this function.

Usage:

```
if (HMPI_Is_host())
{
   printf("I'm the host\n");
}
else
{
   printf("I'm not the host\n");
}
```

Return values: Value of 1 is returned if the process is the member of the group. 0 otherwise.

`HMPI_Is_parent`

Is the calling process the parent process of the group?

Synopsis:

```
unsigned char
HMPI_Is_parent
(
    const HMPI_Group* gid
)
```

Parameters:

`gid` --- handle to the HMPI group of processes.

Description: This routine returns **`true`** if the process calling this routine is the parent of the group represented by the handle **`gid`** otherwise **`false`**. Only processes that are members of the group represented by the handle **`gid`** can call this routine.

Usage:

```
    HMPI_Group* gid;

    if (HMPI_Is_parent(gid))
    {
       printf("I'm the parent of the group gid\n");
    }
    else
    {
       printf("I'm not the parent of the group gid\n");
    }
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group represented by the handle **`gid`**.

HMPI_Is_member

Am I a member of the group?

Synopsis:

```
    unsigned char
    HMPI_Is_member
    (
        const HMPI_Group* gid
    )
```

Parameters:

`gid` --- handle to the HMPI group of processes.

Description: This function returns **`true`** if the process calling this routine is the member of the group represented by the handle **`gid`** otherwise **`false`**. Only processes that are members of the group represented by the handle **`gid`** can call this routine.

Usage:

```
HMPI_Group* gid;

if (HMPI_Is_member(gid))
{
   printf(“I’m a member of the group gid\n”);
}
else
{
   printf(“I’m not a member of the group gid\n”);
}
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group represented by the handle **`gid`**.

HMPI_Is_free

Am I a member of the predefined group **`HMPI_FREE_GROUP`**?

Synopsis:

```
unsigned char
HMPI_Is_free()
```

Description: This routine returns **`true`** if the process is free and is member of the predefined group **`HMPI_FREE_GROUP`** and **`false`** otherwise. Any process can call this function.

Usage:

```
if (HMPI_Is_free())
{
   printf(“I’m a free process and member of”
          “ HMPI_FREE_GROUP \n”);
}
else
{
   printf(“I’m not a free process and not a member of”
          “ HMPI_FREE_GROUP \n”);
}
```

Return values: Value of 1 is returned if the process is not the member of any other group other than **`HMPI_FREE_GROUP`**. 0 otherwise.

HMPI_Get_comm

Returns an MPI communicator with communication group of MPI processes

Synopsis:

```
const MPI_Comm*
HMPI_Get_comm
(
    const HMPI_Group* gid
)
```

Parameters:

`gid` --- handle to the HMPI group of processes.

Description: This routine returns an MPI communicator with communication group of MPI processes defined by **`gid`**. This is a local operation not requiring inter-process communication. Application programmers can use this communicator to call the standard MPI communication routines during the execution of the parallel algorithm. This communicator can safely be used in other MPI routines.

Usage:

```
HMPI_Group* gid;
MPI_Comm* comm;

if (HMPI_Is_member(gid))
{
   comm = HMPI_Get_comm(gid);
   if (comm == NULL)
   {
      //error
   }
}
```

Return values: This call returns **`NULL`** if the process is not a member of the group represented by the handle **`gid`**.

`HMPI_Group_create`

Create an HMPI group of processes

Synopsis:

```
int
HMPI_Group_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    void* model_parameters,
    int paramc
)
```

Parameters:

`gid` --- handle to the HMPI group of processes. This is an output parameter.
`model` --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
`model_parameters` --- parameters of the performance model (input parameter)
`paramc` --- number of parameters of the performance model (input parameter)

Description: This routine tries to create a group that would execute the heterogeneous algorithm faster than any other group of processes. In HMPI, groups are not absolutely independent on each other. Every newly created group has exactly one process shared with already existing groups. That process is called a *parent* of this newly created group, and is the connecting link, through which results of computations are passed if the group ceases to exist. **`HMPI_Group_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HMPI group.

Usage:

```
HMPI_Group gid1, gid2, gid3;

int modelp[1] = {5};
unsigned char is_parent_of_nid2 = 0;
unsigned char is_parent_of_nid3 = 0;

// The parent used in the creation of abstract network
// gid1 is the host
if (HMPI_Is_member(HMPI_HOST_GROUP))
{
   HMPI_Group_create(
       &gid1,
       &HMPI_Model_simple,
       modelp,
       1
   );
}

if (HMPI_Is_free())
{
   HMPI_Group_create(
       &gid1,
        &HMPI_Model_simple,
        NULL,
        0
   );
}

// The parent used in the creation of group gid2 is the
// member of group gid1 whose coordinates are given
// {2}
if (HMPI_Is_member(&gid1))
{
   int numc;
   int** coordinates = (int**)malloc(
                            sizeof(int*)
   );
   int rc = HMPI_Group_coordof(
                 &gid1,
                 &numc,
                 coordinates,
   );
   if ((*coordinates)[0] == 2)
   {
      is_parent_of_nid2 = 1;
   }

   free(coordinates[0]);
   free(coordinates);
}

if (is_parent_of_nid2
     || HMPI_Is_free()
)
{
   HMPI_Group_create(
       &nid2,
       &HMPI_Model_simple,
       modelp,
       1
   );
}

// The parent used in the creation of the group gid3 is
// the member of abstract network nid2 whose
// coordinates are given by {3}
if (HMPI_Is_member(&nid2))
{
   int numc;
```

```
      int** coordinates = (int**)malloc(
                               sizeof(int*)
      );

      int result =  HMPI_Group_coordof(
                        &gid2,
                        &numc,
                        coordinates,
      );

      if ((*coordinates)[0] == 3)
      {
         is_parent_of_gid3 = 1;
      }

      free(coordinates[0]);
      free(coordinates);
   }

   if (is_parent_of_nid3
       || HMPI_Is_free()
   )
   {
      HMPI_Group_create(
          &gid3,
          &HMPI_Model_simple,
          modelp,
          1
      );
   }
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Group_auto_create`

Create an HMPI group of processes with optimal number of processes

Synopsis:

```
int
HMPI_Group_auto_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    void** model_parameters,
    int paramc
)
```

Parameters:

`gid` --- handle to the HMPI group of processes. This is an output parameter.
`model` --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
`model_parameters` --- parameters of the performance model (input parameter)
`paramc` --- number of parameters of the performance model (input parameter)

Description: This routine allows application programmers not to bother about finding the optimal number of processes that can execute the parallel application. They can specify only the rest of the parameters thus leaving the detection of the optimal number of processes to the HMPI runtime system. **`HMPI_Group_auto_create`** is a collective operation and must be called by the parent and all the processes, which are not members of any HMPI group.

The parameters **`model_parameters`** and **`param_count`** are input as well as return parameters. User fills only the input-specific part of the parameter **`model_parameters`** and ignores the return parameters specifying the number of processes to be involved in executing the algorithm and their performances. The parameter **`param_count`** passed to the call of the function **`HMPI_Group_auto_create`** represents the number of parameters in the input-specific part of the parameter **`model_parameters`** and on return, it contains the number of parameters in the input-specific part of the parameter **`model_parameters`** plus the number of parameters containing the number of processes to be involved in executing the algorithm and their performances.

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

HMPI_Group_heuristic_auto_create

Uses user-supplied heuristics to create an HMPI group of processes with optimal number of processes

Synopsis:

```
 typedef int (*HMPI_Heuristic_function)(
        int np, int *dp, void *modelp, int paramc);
int
HMPI_Group_heuristic_auto_create
(
    HMPI_Group* gid,
    const HMPI_Model* model,
    HMPI_Heuristic_function hfunc,
    void** model_parameters,
    int paramc
)
```

Parameters:

`gid` --- handle to the HMPI group of processes. This is an output parameter.
`model` --- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
`hfunc` --- User-supplied heuristic function (input parameter)
`model_parameters` --- parameters of the performance model (input parameter)
`paramc` --- number of parameters of the performance model (input parameter)

Description: This routine has the same functionality as **`HMPI_Group_auto_create`** except that it allows application programmers to supply heuristics that minimize the number of process arrangements evaluated.

Application programmers provide the heuristic function **`hfunc`**. The input parameter **`np`** is the number of dimensions in the process arrangement. The input parameter **`dp`** is an integer array of size **`np`** containing the number of processes in each dimension of the process arrangement. The input parameters **`modelp`** and **`paramc`** are the parameters supplied to the performance model. The function **`HMPI_Group_heuristic_auto_create`** evaluates a process arrangement only if the heuristic function **`hfunc`** returns **true**.

A simple heuristic function is shown below, which returns a value **true** only if the process arrangement is a square grid.

```
int Square_grid_only(
    int np, int *dp, void *modelp, int paramc){
    if ((np == 2) && (dp[0] == dp[1]))
       return true;
    return false;
}
```

The function evaluates process arrangements that are square grids only if this heuristic function is provided as an input.

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Group_free`

Free an HMPI group of processes

Synopsis:

```
int
HMPI_Group_free
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HMPI group of processes

Description: This routine deallocates the resources associated with a group object **gid**. **HMPI_Group_free** is a collective operation and must be called by all the processes, which are members of the HMPI group **gid**.

Usage:

```
HMPI_Group gid;
if (HMPI_Is_member(&gid))
{
   HMPI_Group_free(&gid);
}
```

Return values: **HMPI_SUCCESS** on success and an error in case of failure.

3.3 HMPI Runtime updation Functions

HMPI_Recon

Updates the estimation of processor performances dynamically

Synopsis:

```
typedef void (*HMPI_Benchmark_function)(
        const void*, int, void*);

int
HMPI_Recon
(
    HMPI_Benchmark_function func,
    const void* input_p,
    int num_of_parameters,
    void* output_p
)
```

Parameters:

func --- Benchmark user function executed by all the physical processors.
input_p --- Input parameters to the user function.
num_of_parameters --- Number of input parameters to the user function.

`output_p` --- Return parameter on the execution of the user function.

Description: All the processors execute the benchmark function **`func`** in parallel, and the time elapsed by each of the processors to execute the code is used to refresh the estimation of its speed. This is a collective operation and must be called by all the processors in the group associated with the predefined communication universe **HMPI_PROC_WORLD** of HMPI.

This routine allows updating the estimation of processor performances dynamically, at runtime, just before using the estimation by the programming system. It is especially important if computers, executing the HMPI program, are used for other computations as well. In that case, the real performance of processors can dynamically change dependent on the external computations. The use of this routine allows writing parallel programs sensitive to such dynamic variation of the workload of the underlying computer system.

Usage:

```
double Perf_func (
       double l, double w, double h, double delta)
{
   double m,x,y,z;
   for (m = 0.0, x = 0.0; x < l; x += delta)
       for (y = 0.; y < w; y += delta)
           for (z = 0.; z < h; z += delta)
               m += XYZ_func(x,y,z);
   return m * delta * delta * delta;
}

void Benchmark_function
(
   const void* input_p,
   int num_of_p,
   void* output_p
)
{
   double* params = (double*)input_p;
   double result =   Perf_func(
                         params[0],
                         params[1],
                         params[2],
                         params[3]
   );
   *(double*)(output_p) = result;
   return;
}

// All members of group HMPI_PROC_WORLD_GROUP must call
// this function
```

```
    if (HMPI_Is_member(HMPI_PROC_WORLD_GROUP))
    {
       double output_p;
       int rc = HMPI_Recon(
                    Benchmark_function,
                    input_p,
                    4,
                    &output_p
       );

       if (rc != HMPI_SUCCESS)
       {
          //An error has occurred
       }
    }
```

Return values: `HMPI_SUCCESS` on success and an error in case of failure.

3.4 HMPI Estimation Functions

`HMPI_Timeof`

Predict the total time of execution of the algorithm on the underlying hardware without its real execution

Synopsis:

```
   double
   HMPI_Timeof
   (
       const HMPI_Model* model,
       void* model_parameters,
       int paramc
   )
```

Parameters:

`model`--- handle that encapsulates all the features of the performance model in the form of a set of functions generated by the compiler from the description of the performance model (input parameter)
`model_parameters` --- parameters of the performance model (input parameter)
`paramc` --- number of parameters of the performance model (input parameter)

Description: This routine allows application programmers to predict the total time of execution of the algorithm on the underlying hardware without its real execution. This function allows the application programmers to write such a parallel application that can follow different parallel algorithms to solve the same problem, making choice at runtime depending on the particular executing network and its actual performance. This is a local operation that can be called by any process, which is a member of the group associated with the predefined communication universe **`HMPI_COMM_WORLD`** of HMPI.

`HMPI_Timeof` can thus be used to estimate the execution time on HNOCs for each possible set of model parameters **`model_parameters`**. Application programmers can use this function creatively to design best possible heuristics for the set of parameters. Depending on the time estimated for each set, the optimal values of the parameters are determined. These values are then passed to the performance model during the actual creation of the group of processes using the function **`HMPI_Group_create`**.

Usage:

```
algorithm bcast(int p, int n, int ITER, int rooot) {
   coord I=p;
   node {
      I>=0: bench*1;
   };
   link {
      I>=0&&I!=rooot: length*(n*n*ITER*sizeof(double))
                      [rooot]->[I];
   };
   parent[0];
   scheme {
      int i, k;
      for (k = 0; k < ITER; k++)
          for (i = 0; i < p; i++)
             if (i != rooot)
                (100/ITER)%%[rooot]->[i];
   };
};

int main() {
    int p;
    HMPI_Group gid;
    …
    p = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
    if (HMPI_Is_host()) {
       int param_count = 4;
       int model_params[4] = {
           p,
           N,
           ITER,
           root
       };
       double time;
       time = HMPI_Timeof(
                 &HMPI_Model_bcast,
                   &model_params,
                   param_count
          );
          time = (double)time/(double)ITER;
          printf("Number of bytes broadcast = %d,
                  time=%0.9f\n", N*N*8, time);
      }
   }
```

3.5 HMPI Processor Information Functions

`HMPI_Get_number_of_processors`

Returns the number of physical processors of the underlying distributed memory machine

Synopsis:

```
int
HMPI_Get_number_of_processors()
```

Description: This routine returns the number of physical processors of the underlying distributed memory machine. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HMPI.

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group **`HMPI_COMM_WORLD_GROUP`**. **`HMPI_SUCCESS`** is returned on success.

`HMPI_Get_processors_info`

Returns the relative performances of the physical processors of the underlying distributed memory machine

Synopsis:

```
int
HMPI_Get_processors_info
(
    double* relative_performances
)
```

Parameters:

`Relative_performances` --- Output parameter containing the relative performances of the physical processors of the underlying distributed memory machine

Description: This routine returns the relative performances of the physical processors of the underlying distributed memory machine. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HMPI.

Usage:

```
int p = HMPI_Get_number_of_processors();
double speeds = (double*)malloc(
                         sizeof(double)
                         *
                         p
);

int rc = HMPI_Get_processors_info(
             speeds
);

if (rc != HMPI_SUCCESS)
{
   //An error has occurred
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group **HMPI_COMM_WORLD_GROUP**. **HMPI_SUCCESS** is returned on success.

HMPI_Get_processes_info

Returns the relative performances of the processes running on the physical processors of the underlying distributed memory machine

Synopsis:

```
int
HMPI_Get_processes_info
(
    double* relative_performances
)
```

Parameters:

Relative_performances --- Output parameter containing the relative performances of the processes running on the physical processors of the underlying distributed memory machine

Description: This routine returns the relative performances of the processes running on the physical processors of the underlying distributed memory machine. This is a collective operation and must be called by all the processes in the group associated with the predefined communication universe **HMPI_COMM_WORLD** of HMPI.

Usage:

```
int p = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
double speeds = (double*)malloc(
                         sizeof(double)
                         *
                         p
);

int rc = HMPI_Get_processes_info(
             speeds
);

if (rc != HMPI_SUCCESS)
{
   //An error has occurred
}
```

Return values: Error code **`HMPI_UNDEFINED`** is returned if the process is not the member of the group **`HMPI_COMM_WORLD_GROUP`**. **`HMPI_SUCCESS`** is returned on success.

HMPI_Group_performances

Returns the relative performances of the processes in a group

Synopsis:

```
int
HMPI_Group_performances
(
    const HMPI_Group* gid,
    double* relative_performances
)
```

Parameters:

`gid` --- handle to the HMPI group of processes
`Relative_performances` --- Output parameter containing the relative performances of the processes in the group represented by the handle **`gid`**

Description: This routine returns the relative performances of the processes in the group represented by the handle **gid**. This is a collective operation and must be called by all the processes in the group given by the handle **gid**.

Usage:

```
HMPI_Group gid;

if (HMPI_Is_member(gid))
{
   int p = HMPI_Group_size(&gid);
   double speeds = (double*)malloc(
                         sizeof(double)
                         *
                         p
   );

   int rc = HMPI_Group_performances(
                 gid,
                 speeds
   );

   if (rc != HMPI_SUCCESS)
   {
      //An error has occurred
   }
}
```

Return values: Error code **HMPI_UNDEFINED** is returned if the process is not the member of the group given by the handle **gid**. **HMPI_SUCCESS** is returned on success.

3.6 HMPI Synchronization Functions

HMPI_Barrier

Barrier for the members of the group

Synopsis:

```
int HMPI_Barrier
(
    const HMPI_Group* gid
)
```

Parameters:

gid --- handle to the HMPI group of processes

Description: Has same functionality as **`MPI_Barrier`**. This is a collective operation and must be called by all the processes in the group given by the handle **`gid`**.

Usage:

```
    HMPI_Group gid;

    if (HMPI_Is_member(&gid))
    {
       HMPI_Barrier(&gid);
    }
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Notify_free_processes`

Notify free processes to leave the waiting point

Synopsis:

```
int HMPI_Notify_free_processes()
```

Description: This must be called by only the host-process. It sends a command to the dispatcher to signal the free processes to leave the waiting point.

Usage:

```
    HMPI_Group gid;

    if (HMPI_Is_host())
    {
       HMPI_Notify_free_processes();
    }
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Wait_free_processes`

Waiting point for free processes waiting for commands for group destruction

Synopsis:

```
int HMPI_Wait_free_processes()
```

Description: This must be called by all the free processes. All the free processes wait in this call for commands from dispatcher on group destruction.

Usage:

```
    if (HMPI_Is_free())
    {
       HMPI_Wait_free_processes();
    }
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

`HMPI_Host_rendezvous`

Allows rendezvous with the host-process

Synopsis:

```
int HMPI_Host_rendezvous(int count)
```

Description: This function allows rendezvous with the host-process. Any process, which is the member of the group **`HMPI_COMM_WORLD_GROUP`**, and the host-process must call this function.

Parameters:

`count` --- Number of processes rendezvous with the host-process

Usage:

```
    HMPI_Group gid;
    // A parent of a group can rendezvous with the host
    if (HMPI_Is_parent(&gid) || HMPI_Is_host())
    {
       HMPI_Host_rendezvous(1);
    }

    // A whole group can rendezvous with the host
    if (HMPI_Is_member(&gid) || HMPI_Is_host())
    {
       HMPI_Host_rendezvous(HMPI_Group_size(&gid));
    }
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

3.7 HMPI Debugging and Version Functions

`HMPI_Printf`

Print formatted strings to the host processor.

Synopsis:

```
int HMPI_Printf
(
    const char* format,
    ...
)
```

Parameters:

`format` --- Format string in printf-fashion.

Description: Prints formatted strings to standard output on the virtual host processor from any virtual processor of the computing space. Any process can call this function.

Usage:

```
    HMPI_Group gid;

    if (HMPI_Is_member(&gid))
    {
       HMPI_Printf(
          "Hello, My node rank is %d, My Globalrank “
          “is %d\n ",
          HMPI_Group_rank(&nid),
          HMPI_Group_rank(HMPI_COMM_WORLD_GROUP)
       );
    }
```

Return values: **`HMPI_SUCCESS`** on success and an error in case of failure.

HMPI_Strerror

Return a string associated with error code.

Synopsis:

```
int
HMPI_Strerror
(
    int errnum,
    char* message
)
```

Parameters:

`errnum` --- Error code from any HMPI routine call.

`message` --- Output parameter. Error message associated with the error code. The message must represent storage that is at least **`HMPI_MAX_ERROR_STRING`** characters long.

Description: An error message string corresponding to the error number **`errnum`** is returned in **`message`**. Any process can call this function.

Usage:

```
    char message[HMPI_MAX_ERROR_STRING];

    int rc = HMPI_Init(
                 argc,
                 argv
    );

    if (rc != HMPI_SUCCESS)
    {
       HMPI_Strerror(
           rc,
           message
       );

       HMPI_Printf(
        "Error during HMPI initialization. Reason is %s\n",
        message
       );
    }
```

Return values: **`HMPI_SUCCESS`** on success and error on failure.

HMPI_Debug

Turn the diagnostics on/off.

Synopsis:

```
int
HMPI_Debug
(
    int yesno
)
```

Parameters:

`yesno` --- yes (1) or no (0)

Description: Produces detailed diagnostics. Any process can call this function. This is the only function apart from **`HMPI_Get_version`** that can be called before **`HMPI_Init`** or after **`HMPI_Finalize`**.

HMPI_Get_version

Returns the version of the HMPI in the format x.y

Synopsis:

```
int
HMPI_Get_version
(
    int *version,
    int *sub_version
)
```

Parameters:

`version` --- Major version
`sub_version` --- Minor version

Description: Returns the version of HMPI. Any process can call this function. This is one of the few functions that can be called before **`HMPI_Init`** or after **`HMPI_Finalize`**.

Usage:

```
    int version, sub_version;
    HMPI_Get_version(&version, &sub_version);
```

4 Heterogeneous Data Partitioning Interface (HDPI)

The core of scientific, engineering or business applications is the processing of some mathematical objects that are used in modeling corresponding real-life problems. In particular, partitioning of such mathematical objects is a core of any data parallel algorithm. Our analysis of various scientific, engineering and business domains resulted in the following short list of mathematical objects commonly used in parallel and distributed algorithms: **sets** (ordered and non-ordered), **dense matrices** (and multidimensional arrangements), **graphs**, and **trees**.

Based on this classification, we suggest an API for partitioning mathematical objects commonly used in scientific and engineering domains for solving problems on networks of heterogeneous computers. These interfaces allow the application programmers to specify simple and basic partitioning criteria in the form of parameters and functions to partition their mathematical objects. These partitioning interfaces are designed to be used along with various programming tools for parallel and distributed computing on heterogeneous networks.

4.1 Sets

`Partition_unordered_set`

Partition a non-ordered set

Synopsis:

```
typedef double (*User_defined_metric)(
      int p, const double *speeds, const int *actual,
      const int *ideal)

int Partition_unordered_set (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int n,
    const int *w, int type_of_metric,
    User_defined_metric umf, double *metric, int *np)
```

Description: This routine partitions a set into **`p`** disjoint partitions.

Return values: **`0`** on success and **`-1`** in case of failure.

`Partition_ordered_set`

Partition a well-ordered set

Synopsis:

```
int Partition_ordered_set (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int n,
    const int *w, int processor_reordering,
    int type_of_metric, User_defined_metric umf,
    double *metric, int *np)
```

Description: This routine partitions a well-ordered set into **`p`** disjoint contiguous partitions.

Parameters:

Parameter **`p`** is the number of partitions of the set. Parameters **`speeds`** and **`psizes`** specify speeds of processors for **`pn`** different problem sizes. These parameters are 1D arrays of size **`p×pn`** logically representing 2D arrays of shape **`[p][pn]`**. The speed of the **`i`**-th processor for **`j`**-th problem size is given by the **`[i][j]`**-th element of **`speeds`** with the problem size itself given by the **`[i][j]`**-th element of **`psizes`**. Parameter **`mlimits`** gives the maximum number of elements that each processor can hold.

Parameter **`n`** is the number of elements in the set, and parameter **`w`** is the weights of its elements.

Parameter **`type_of_metric`** specifies which metric should be used to determine the quality of the partitioning. If **`type_of_metric`** is **`USER_SPECIFIED`**, then the user provides a metric function **`umf`**, which is used to calculate the quality of the partitioning. If **`type_of_metric`** is **`SYSTEM_DEFINED`**, the system-defined metric is used.

The output parameter **`metric`** gives the quality of the partitioning, which is the deviation of the partitioning achieved from the ideal partitioning satisfying the partitioning criteria. If the output parameter **`metric`** is set to **`NULL`**, then the calculation of metric is ignored.

If **`w`** is not **`NULL`** and the set is well ordered, then the user needs to specify if the implementations of this operation may reorder the processors before partitioning (Boolean parameter **`processor_reordering`** is used to do it). One typical reordering is to order the processors in the decreasing order of their speeds.

Return values: **`0`** on success and **`-1`** in case of failure.

`Get_set_processor`

For an ordered set, returns the processor owning the set element at index **i**

Synopsis:

```
int Get_set_processor (
     int i, int n, int p, int processor_reordering,
     const int *np)
```

Return values: **`-1`** in case of failure.

`Get_my_partition`

For a set, returns the number of elements allocated to processor **i**

Synopsis:

```
int Get_my_partition (
     int i, int p, const double *speeds, int n)
```

Return values: **`-1`** in case of failure.

4.2 Dense Matrices

`Partition_matrix_2d`

Partition a matrix amongst processors arranged in a 2D grid

Synopsis:

```
int Partition_matrix_2d (
    int p, int q,
    int pn, const double *speeds, const int *psizes,
    const int *mlimits, int m, int n,
   int type_of_distribution, int *w, int *h, int *trow,
   int *tcol, int *ci, int *cj )
```

Parameters:

The parameter **`p`** is the number of processors along the row of the processor grid. The parameter **`q`** is the number of processors along the column of the processor grid.

Parameters **`speeds`** and **`psizes`** specify speeds of processors for **`pn`** different problem sizes. These parameters are 1D arrays of size **`p×q×pn`** logically representing arrays of shape **`[p][q][pn]`**. The speed of the **`(i, j)`**-th processor for **`k`**-th problem size is given by the **`[i][j][k]`**-th element of **`speeds`** with the problem size itself given by the **`[i][j][k]`**-th element of **`psizes`**. Parameter **`mlimits`** gives the maximum number of elements that each processor can hold.

The parameters **`m`** and **`n`** are the sizes of the generalized block along the row and the column.

The input parameter **`type_of_distribution`** specifies if the distribution is **`CARTESIAN`**, **`ROW-BASED`**, and **`COLUMN-BASED`**.

Output parameter **`w`** gives the widths of the rectangles of the generalized block assigned to different processors. This parameter is an array of size **`p`**×**`q`**.

Output parameter **`h`** gives the heights of rectangles of the generalized block assigned to different processors. This parameter is an array of size **`p`**×**`q`**×**`p`**×**`q`** logically representing array of shape **`[p][q][p][q]`**.

Output parameter **`trow`** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first row of the generalized block. This parameter is an array of size **`p`**×**`q`**.

Output parameter **`tcol`** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first column of the generalized block. This parameter is an array of size **`p`**×**`q`**.

Output parameters **`ci`**, and **`cj`** are each an array of size **`m`**×**`n`**. The coordinates of the processor in its processor grid to which the matrix element at row **`i`** and column **`j`** of the generalized block is assigned is given by **`ci[i×n+j]`**, and **`cj[i×n+j]`** respectively. If the application programmer sets these parameters to **`NULL`**, then these parameters are ignored.

Description: This routine partitions a matrix into **`p`** disjoint partitions amongst processors arranged in a 2D grid.

Return values: **`0`** on success and **`-1`** in case of failure.

`Partition_matrix_1d_dp`

Partition a matrix amongst processors arranged in a linear array

Synopsis:

```
int Partition_matrix_1d_dp(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, DP_function dpf,
    int type_of_distribution,
    int *w, int *h, int *trow, int *tcol, int *c)
```

Parameters:

The parameter **`p`** is the number of number of disjoint rectangles the matrix is partitioned into. Parameters **`speeds`** and **`psizes`** specify speeds of processors for **`pn`** different problem sizes. These parameters are 1D arrays of size **`p×pn`** logically representing 2D arrays of shape **`[p][pn]`**. The speed of the **`i`**-th processor for **`j`**-th problem size is given by the **`[i][j]`**-th element of **`speeds`** with the problem size itself given by the **`[i][j]`**-th element of **`psizes`**. Parameter **`mlimits`** gives the maximum number of elements that each processor can hold.

The parameters **`m`** and **`n`** are the sizes of the generalized block along the row and the column.

The input parameter **`type_of_distribution`** specifies if the distribution is **`ROW-BASED`** **`o`**r **`COLUMN-BASED`**.

Output parameter **`w`** gives the widths of the rectangles of the generalized block assigned to different processors. This parameter is an array of size **`p`**. Output parameter **`h`** gives the heights of rectangles of the generalized block assigned to different processors. This parameter is an array of size **`p`**×**`p`**. Output parameter **`trow`** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first row of the generalized block. This parameter is an array of size **`p`**. Output parameter **`tcol`** gives the top leftmost point of the rectangles of the generalized block assigned to different processors from the first column of the generalized block. This parameter is an array of size **`p`**.

Output parameter **`c`** is an array of size **`m`**×**`n`**. The coordinates of the processor in its processor array to which the matrix element at row **`i`** and column **`j`** of the generalized block is assigned is given by **`c[i×n+j]`**. If the user sets these parameters to **`NULL`**, then these parameters are ignored.

Description: This routine partitions a matrix into **`p`** disjoint partitions amongst processors arranged in a linear array.

Return values: **`0`** on success and **`-1`** in case of failure.

`Partition_matrix_1d_iterative`

Partition a matrix amongst processors arranged in a linear array

Synopsis:

```
int Partition_matrix_1d_iterative(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, Iterative_function cf,
    int *w, int *h, int *trow, int *tcol, int *c)
```

Parameters:

Application programmers provide a cost function **`cf`** that tests the optimality of a partition from a finite set of partitions. The initial partition in this finite set of partitions is obtained using a problem-specific strategy. The cost function **`cf`** is called iteratively for each of the partitions in the subset of partitions. The return value of this function gives an optimality value. At each step of the iteration, the optimality value is compared to the lower bound of the optimal solution to the optimization problem. Application programmers specify a function **`lb`**, which is used to calculate the lower bound of their optimization problem. The iteration stops when the function returns an optimality value less than or equal to the lower bound or a negative return value indicating that the partitioning cannot be improved and that the current partition is optimal.

Description: Partitions a matrix into **`p`** disjoint partitions amongst processors arranged in a linear array.

Return values: **`0`** on success and **`-1`** in case of failure.

`Partition_matrix_1d_refining`

Partition a matrix amongst processors arranged in a linear array

Synopsis:

```
int Partition_matrix_1d_refining(
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int m, int n,
    Get_lower_bound lb, Refining_function cf,
    int *w, int *h, int *trow, int *tcol, int *c)
```

Parameters:

Application programmers provide a refinement function **`rf`** that refines an old partition giving a new better partition. A negative return value of this function suggests that the old partition cannot be refined further. This function is iteratively called. The partition for the first call of this refining function is obtained using a problem-specific strategy. Application programmers specify a function **`lb`**, which is used to calculate the lower bound of their optimization problem. The iteration stops when the refinement function **`rf`** returns an optimality value less than or equal to the lower bound indicating that the current partition is optimal.

Description: Partitions a matrix into **`p`** disjoint partitions amongst processors arranged in a linear array.

Return values: **`0`** on success and **`-1`** in case of failure.

`Get_matrix_processor`

Returns the coordinates (**i**,**j**) of the processor owning the matrix element at row **r** and column **c**

Synopsis:

```
typedef struct {int i; int j;} Processor;
int Get_matrix_processor(
    int r, int c, int p, int q, int *w, int *h, int *trow,
    int *tcol, int type_of_distribution, Processor *root)
```

Return values: **`0`** on success and **`-1`** in case of failure.

`Get_my_width`

Returns the width of the rectangle owned by the processor with coordinates (**i**,**j**)

Synopsis:

```
int Get_my_width(
    int i, int j, int p, int q, const double *speeds,
    int type_of_distribution, int m, int n)
```

Description: Currently only applicable to two-dimensional processor arrangements.

Return values: **`-1`** in case of failure.

`Get_my_height`

Returns the height of the rectangle owned by the processor with coordinates (**i**,**j**)

Synopsis:

```
int Get_my_height(
    int i, int j, int p, int q, const double *speeds,
    int type_of_distribution, int m, int n)
```

Description: Currently only applicable to two-dimensional processor arrangements.

Return values: **`-1`** in case of failure.

`Get_diagonal`

Obtain the number of elements owned by the processor with coordinates (**i**,**j**) on the diagonal of the matrix

Synopsis:

```
int Get_diagonal(
    int i, int j, int p, int q, int *w, int *h, int *trow,
    int *tcol)
```

Description: Currently only applicable to dense square matrices and two-dimensional processor arrangements.

Return values: **`-1`** in case of failure.

`Get_my_elements`

Obtain the number of elements owned by the processor with coordinates (**i**,**j**) in the upper or lower half of the matrix including the diagonal elements

Synopsis:

```
int Get_my_elements(
    int n, int g, int i, int j, int p, int q, int *w, int *h,
    int *trow, int *tcol, int type_of_distribution,
    char upper_or_lower)
```

Description: Currently only applicable to dense square matrices and two-dimensional processor arrangements.

Return values: **`-1`** in case of failure.

`Get_my_kk_elements`

Obtain the number of elements owned by the processor with coordinates (**i**,**j**) in the upper or lower half of the matrix starting from (**k**,**k**) including the diagonal elements

Synopsis:

```
int Get_my_kk_elements(
    int n, int g, int k, int i, int j, int p, int q, int *w,
    int *h, int *trow, int *tcol, int type_of_distribution,
    char upper_or_lower)
```

Description: Currently only applicable to dense square matrices and two-dimensional processor arrangements.

Return values: **`-1`** in case of failure.

4.3 Graphs

`Partition_graph`

Partition a graph

Synopsis:

```
int Partition_graph (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits, int n, int m,
    const int *vwgt, const int *xadj,
    const int *adjacency, const int *adjwgt,
    int nopts, const int *options, int *vp, int *edgecut)
```

Parameters:

Parameter **`p`** is the number of partitions of the graph. Parameters **`speeds`** and **`psizes`** specify speeds of processors for **`pn`** different problem sizes. These parameters are 1D arrays of size **`p×pn`** logically representing 2D arrays of shape **`[p][pn]`**. The speed of the **`i`**-th processor for **`j`**-th problem size is given by the **`[i][j]`**-th element of **`speeds`** with the problem size itself given by the **`[i][j]`**-th element of **`psizes`**. Parameter **`mlimits`** gives the maximum number of elements that each processor can hold.

The parameters **`n`** and **`m`** are the number of vertices and edges in the graph. The parameters **`vwgt`** and **`adjwgt`** are the weights of vertices and edges of the graph. In the case in which the graph is unweighted (i.e., all vertices and/or edges have the same weight), then either or both of the arrays **`vwgt`** and **`adjwgt`** can be set to **`NULL`**. The parameters **`vwgt`** is of size **`n`**. The parameter **`adjwgt`** is of size **`2m`** because every edge is listed twice (i.e., as (*v*, *u*) and (*u*, *v*)).

The parameters **`xadj`** and **`adjacency`** specify the adjacency structure of the graph represented by the compressed storage format (CSR). The adjacency structure of the graph is stored as follows. The adjacency list of vertex **`i`** is stored in **`adjacency`** starting at index **`xadj[i]`** and ending at but not including **`xadj[i+1]`**. The adjacency lists for each vertex are stored consecutively in the array **`adjacency`**.

The parameter **`options`** is an array of size **`nopts`** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **`vp`** is an array of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which vertex **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

Description: This routine partitions a graph into **`p`** disjoint partitions.

Return values: **`0`** on success and **`-1`** in case of failure.

Partition_bipartite_graph

Partition a bipartite graph

Synopsis:

```
  int Partition_bipartite_graph (
      int p, int pn, const double *speeds,
       const int *psizes, const int *mlimits,
       int n, int m, const int *vtype, const int *vwgt,
       const int *xadj, const int *adjacency,
       const int *adjwgt, int type_of_partitioning,
       int nopts, const int *options, int *vp, int *edgecut)
```

Parameters:

The meaning of the parameters **`p`**, **`pn`**, **`speeds`**, **`psizes`**, **`mlimits`**, **`n`**, **`m`**, **`vwgt`**, **`adjwgt`**, **`xadj`**, **`adjacency`** is identical to meaning of the corresponding parameters of **`Partition_graph`**.

The parameter **`vtype`** specifies the type of vertex. The only values allowed are 0 and 1 representing the two disjoint subsets the bipartite graph is composed of.

The parameter **`type_of_partitioning`** specifies whether the partitioning of subsets is done separately or not. It can take only one of the values **`PARTITION_SUBSET`** and **`PARTITION_OTHER`**.

The parameter **`options`** is an array of size **`nopts`** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which vertex **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

Description: This routine partitions a bipartite graph into **`p`** disjoint partitions.

Return values: **`0`** on success and **`-1`** in case of failure.

`Partition_hypergraph`

Partition a hypergraph

Synopsis:

```
int Partition_hypergraph (
    int p, int pn, const double *speeds,
    const int *psizes, const int *mlimits,
    int nv, int nedges, const int *vwgt, const int *hptr,
    const int *hind, const int *hwgt, int *vp,
    int nopts, const int *options, int *edgecut)
```

Parameters:

The meaning of the parameters **`p`**, **`pn`**, **`speeds`**, **`psizes`**, and **`mlimits`** is identical to meaning of the corresponding parameters of **`Partition_graph`**.

The parameters `nv` and `nedges` are the number of vertices and number of hyperedges in the hypergraph.

The parameters `vwgt` is an array of size `nv` that stores the weights of the vertices and `hwgt` is an array of size `nedges` that stores the weights of hyperedges of the graph. If the vertices in the hypergraph are unweighted, then `vwgt` can be `NULL`. If the hyperedges in the hypergraph are unweighted, then `hwgt` can be `NULL`.

The parameter **`hptr`** is an array of size **`nedges`**`+1` and is an index into **`hind`** that stores the actual hyperedges. Each hyperedge stores the sequence of the vertices that it spans, in consecutive locations in **`hind`**. Specifically, **`i`**-th hyperedge is stored starting at location **`hind[hptr[i]]`** up to but not including **`hind[hptr[i+1]]`**.

The parameter **`options`** is an array of size **`nopts`** containing the options for the various phases of the partitioning algorithms employed in partitioning the graph. These options allow integration of third party implementations, which provide their own partitioning schemes.

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which vertex **`i`** belongs to. The parameter **`edgecut`** contains the number of hyperedges that are cut by the partitioning.

Description: This routine partitions a hypergraph into **`p`** disjoint partitions.

Return values: **`0`** on success and **`-1`** in case of failure.

4.4 Trees

`Partition_tree`

Partition a tree

Synopsis:

```
  int Partition_tree (
      int p, int pn, const double *speeds,
      const int *psizes, const int *mlimits,
      int n, int nedges, const int *nwgt, const int *xadj,
      const int *adjacency, const int *adjwgt,
      int *vp, int *edgecut)
```

Parameters:

The meaning of the parameters **`p`**, **`pn`**, **`speeds`**, **`psizes`**, and **`mlimits`** is identical to meaning of the corresponding parameters of **`Partition_graph`**.

The parameters `n` and `nedges` are the number of vertices and edges in the tree. The parameters `nwgt` is an array of size `n` that stores the weights of the vertices and `adjwgt` is an array of size `nedges` that stores the weights of edges of the tree. If the vertices in the tree are unweighted, then `nwgt` can be `NULL`. If the edges in the tree are unweighted, then `adjwgt` can be `NULL`.

The parameters **`xadj`** and **`adjacency`** specify the adjacency structure of the tree.

The parameter **`vp`** is an array of size of size **`n`** containing the partitions to which the vertices are assigned. Specifically, **`vp[i]`** contains the partition number in which node **`i`** belongs to. The parameter **`edgecut`** contains the number of edges that are cut by the partitioning.

Description: This routine partitions a tree into **`p`** disjoint subtrees.

Return values: **`0`** on success and **`-1`** in case of failure.

5 HMPI Command-line User's Interface

5.1 HMPI Environment

Currently, the HMPI programming environment includes a *compiler*, *run-time support system* (RTS), a *library*, and a *command-line user interface*.

The compiler compiles the description of this performance model to generate a set of functions. The functions make up an algorithm-specific part of the HMPI runtime system.

The library consists of extensions to MPI and Heterogeneous Data Partitioning Interface (HDPI).

HMPI command-line user's interface consists of a number of utilities supporting parallel machines manipulation actions and building of HMPI applications.

5.2 Virtual Parallel Machine

Please refer to the mpC command-line user's interface guide on how to write a VPM description file and the VPM manipulation utilities:

- "**`mpccreate`**" to create a VPM;
- "**`mpcopen`**" to create a VPM;
- "**`mpcclose`**" to close a VPM;
- "**`mpcdel`**" to remove a VPM;

```
nettype grid(int p, int q) {
   coord I=p, J=q;
};
```

Figure A.1: Specification of a simple performance model in the HMPI's performance definition language. The performance model definition is in the file "**`grid.mpc`**".

5.3 Building and Running HMPI Application

Please refer to the mpC command-line user's interface guide on utilities that are used to run an mpC/HMPI application on a VPM:

- "**`hmpicc`**" to compile a performance model definition file;
- "**`hmpibcast`**" to make available all the source files to build a executable;
- "**`hmpiload`**" to create a executable;
- "**`hmpirun`**" to execute the target application;

A sample performance model and the HMPI application using the performance model are shown in Figures A.1 and A.2:

Outlined below are steps to build and run a HMPI application.

1). The first step is to describe your Virtual Parallel Machine (VPM). This consists of all the machines being used in your HMPI application. Describe your VPM in a file in the **`$MPCLOCAL/topo`** directory. VPM is opened after successful execution of the command **`mpccreate`**. Consider for example:

```
shell$ cat $MPCLOCAL/topo/vpm_Solmach123_Linuxmach456.vpm

#
# Machines and the number of processes to run on each
# machine
# Number in square brackets indicate the number of
# processors
```

```
#include <math.h>
#include <stdio.h>
#include <sys/time.h>
#include "grid.c"

int main() {
    int param_count, model_params[2];
    struct timeval start, end;
    gettimeofday(&start, NULL);

    HMPI_Group gid;
    HMPI_Init(argc, argv);
    if (HMPI_Is_host()) {
       int gsize, p, q;
       param_count = 2;
       gsize = HMPI_Group_size(HMPI_COMM_WORLD_GROUP);
       p = q = sqrt(gsize);
       if ((p == 0) && (q == 0))
          p = q = 1;
       model_params[0] = p;
       model_params[1] = q;

       printf("Total number of processes available for computation
              is %d\n", gsize);
       printf("Creating a grid (%d, %d) of processes\n", p, q);
    }
    if (HMPI_Is_host())
       HMPI_Group_create (&gid, &MPC_NetType_grid,
                          model_params, param_count)
    if (HMPI_Is_free())
       HMPI_Group_create (&gid, &MPC_NetType_grid,
                          NULL, 0)
    // Distribute computations using the optimal speeds of processes
    if (HMPI_Is_member(&gid)){
      // computations and communications are performed here
    }
    if (HMPI_Is_member(&gid)) HMPI_Group_free(&gid);
    gettimeofday(&end, NULL);
    if (HMPI_Is_host()) {
       double tstart = start.tv_sec + (start.tv_usec/pow(10, 6));
       double tend = end.tv_sec + (end.tv_usec/pow(10, 6));
       printf("Time taken for group creation(sec)=%f\n",
              tend-tstart);
    }
    HMPI_Finalize(0);
}
```

Figure A.2: A sample HMPI program. The HMPI program is written in the file "T**`est_group_create.c`**".

```
solmach1 2 [2]
solmach2 2 [2]
solmach3 2 [2]
linuxmach4 4 [4]
```

```
linuxmach5 2 [2]
linuxmach6 1 [1]
```

```
shell$ mpccreate vpm_Solmach123_Linuxmach456
```

2). Compile the performance model file.

```
shell$ hmpicc grid.mpc
```

This file is translated into a C file "**`grid.c`**".

3). Broadcast the files to all the machines in the virtual parallel machine.

```
shell$ hmpibcast Test_group_create.c grid.c
```

4). Create the executable.

```
shell$ hmpiload –o Test_group_create Test_group_create.c
```

5). Run the target program.

```
shell$ hmpirun Test_group_create
Total number of processes available for computation is 9
Creating a grid (3, 3) of processes
Time taken for group creation(sec)=0.262353
```

6 HMPI Installation Guide for UNIX

This section provides information for programmers and/or system administrators who want to install HMPI for UNIX.

6.1 System Requirements

The following table describes system requirements for HMPI for UNIX.

Component	Requirement
Operating System	Linux, Solaris, FreeBSD **HMPI** is successfully tested on the following operating systems: **Linux 2.6.5-1.358smp (gcc version 3.3.3 20040412 (Red Hat Linux 3.3.3-7))** **Linux 2.6.8-1.521smp (gcc version 3.3.3 20040412 (Red Hat Linux 3.3.3-7))**

	Linux 2.6.5-1.358 (gcc version 3.3.3 20040412 (Red Hat Linux 3.3.3-7)) **Linux 2.4.18-3 ((gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-110))** **Sun Solaris 5.9 (gcc version 3.4.1)** **FreeBSD 5.2.1-RELEASE (gcc version 3.3.3 [FreeBSD] 20031106)**
C compiler	Any ANSI C compiler
MPI	LAM MPI 6.3.2 or higher MPICH MPI 1.2.0 or higher with chp4 device
mpC	Version 3.0.0 or higher

LAM MPI can be obtained from http://www.lam-mpi.org/
MPICH MPI can be obtained from http://www-unix.mcs.anl.gov/mpi/mpich/
mpC package can be obtained from http://www.ispras.ru/~mpc/

6.2 Contents of HMPI for UNIX Distribution

HMPI for Unix distribution contains the following:

Directory	Contents
README	Copyright information, Contact information
INSTALL	Installation instructions
Makefile	Installation and test of the compiler and the environment
docs	HMPI manual for programmers
man	Manual pages for HMPI API
src	Source code for HMPI
include	Header files
tests	Tests for testing HMPI library
Third_Party_Software	Third party software for graphs
tools	HMPI tools to build executables, clean up HMPI repositories

6.3 Before Installation

6.3.1 Installing MPI

You should have MPI installed on your system. Please make sure that **`mpicc`** and **`mpirun`** scripts are in your **`PATH`** environment variable.

```
...
shell$ export MPIDIR=<...MPI install directory...>
shell$ export PATH=$MPIDIR/bin:$PATH
...
```

6.3.2 Installing mpC

You should have mpC installed on your system. Please refer to the mpC installation guide on the variables to export in the shell startup files.

```
...
shell$ export MPCHOME=<...mpC install directory...>
shell$ export PATH=$MPCHOME/bin:$PATH
...
```

6.3.3 Making rsh/ssh working

If you using **`rsh`**, please make sure that you reach every machine from every other machine with **`rsh`** command by executing **`rsh –n true hostname`**. This command should not hang up.

If you are using **`ssh`**, please follow the instructions below:

Normally, when you use **`ssh`** to connect to a remote host, it will prompt you for your password. However, in order for MPI commands to work properly, you need to be able to execute jobs on remote nodes without typing in a password. In order to do this, you will need to set up RSA (ssh 1.x and 2.x) or DSA (ssh 2.x) authentication.

This text will briefly show you the steps involved in doing this, but the **`ssh`** documentation is authoritative on these matters should be consulted for more information. The first thing that you need to do is generate an DSA key pair to use with **`ssh-keygen`**:

```
shell$ ssh-keygen -t dsa
```

Accept the default value for the file in which to store the key (**`$HOME/.ssh/id_dsa`**) and enter a passphrase for your keypair. You may choose to not enter a passphrase and therefore obviate the need for using the **`ssh-agent`**. However, this weakens the authentication that is possible, because your secret key is [potentially] vulnerable to compromise because it is unencrypted. See the **`ssh`** documentation.

Next, copy the **`$HOME/.ssh/id_dsa.pub`** file generated by **`ssh-keygen`** to **`$HOME/.ssh/authorized_keys`**:

```
shell$ cd $HOME/.ssh
shell$ cp id_dsa.pub authorized_keys
```

In order for DSA authentication to work, you need to have the **$HOME/.ssh** directory in your home directory on all the machines you are running MPI on. If your home directory is on a common filesystem, this is already taken care of. If not, you will need to copy the **$HOME/.ssh** directory to your home directory on all MPI nodes (be sure to do this in a secure manner -- perhaps using the **scp** command), particularly if your secret key is not encrypted).

ssh is very particular about file permissions. Ensure that your home directory on all your machines is set to mode 755, your **$HOME/.ssh** directory is also set to mode 755, and that the following files inside **$HOME/.ssh** have the following permissions:

```
-rw-r--r--  authorized_keys
-rw-------  id_dsa
-rw-r--r--  id_dsa.pub
-rw-r--r--  known_hosts
```

You are now set up to use DSA authentication. However, when you **ssh** to a remote host, you will still be asked for your DSA *passphrase* (as opposed to your normal *password*). This is where the **ssh-agent** program comes in. It allows you to type in your DSA passphrase once, and then have all successive invocations of **ssh** automatically authenticate you against the remote host. To start up the **ssh-agent**, type:

```
shell$ eval `ssh-agent`
```

You will probably want to start the **ssh-agent** before you start X windows, so that all your windows will inherit the environment variables set by this command. Note that some sites invoke **ssh-agent** for each user upon login automatically; be sure to check and see if there is an **ssh-agent** running for you already. Once the **ssh-agent** is running, you can tell it your passphrase by running the **ssh-add** command:

```
shell$ ssh-add $HOME/.ssh/id_dsa
```

At this point, if you **ssh** to a remote host that has the same **$HOME/.ssh** directory as your local one, you should not be prompted for a password. If you are, a common problem is that the permissions in your **$HOME/.ssh** directory are not as they should be.

Note that this text has covered the **ssh** commands in very little detail. Please consult the **ssh** documentation for more information.

6.4 Beginning Installation

Unpack the HMPI distribution, which comes as a tar in the form hmpi-x.y.tar.gz.

To uncompress the file tree use:

```
shell$ gzip -d hmpi-x.y.z.tar.gz
shell$ tar -xvf hmpi-x.y.z.tar
```

where x.y.z stands for the installed version of the HMPI library (say 1.2.1, 2.0.0, or 3.1.1).

The directory 'hmpi-x.y.z' will be created; execute

```
shell$ cd hmpi-x.y.z
```

The Makefile at the global level (hmpi-x.y.z/Makefile) controls the compilation and installation of the HMPI software. It activates subdirectory specific Makefiles.

Export the variable **HMPI_HOME** to point to the installation directory (directory where binaries of HMPI will be installed)

```
shell$ export HMPI_HOME=<...install directory...>
```

To compile all the programs execute:

```
shell$ ./install_hmpi
```

To clean up:

```
shell$ make clean
```

to remove object files and executables from source directories.

6.5 Finishing Installation

On successful installation of HMPI, the following message is displayed:

```
###########################################################
   Installation of HMPI SUCCESSFUL
   export the variable
export HMPI_HOME=/home/cs/manredd/mpC3.x.x/mpcc-
3.x.x/apps/HMPI/dev/HMPI_Linux_2.6.8-1.521smp
   Set the value below in PATH environment variable
/home/cs/manredd/mpC3.x.x/mpcc-
3.x.x/apps/HMPI/dev/HMPI_Linux_2.6.8-1.521smp/bin
###########################################################
```

You should update your shell startup files with the following variables:

...

```
shell$ export HMPI_HOME=<...install directory...>
shell$ export PATH=$HMPI_HOME/bin:$PATH
...
```

6.6 Contents of HMPI Installation

HMPI installation contains the following:

Directory	Contents
bin	Binaries **hmpicc**, **hmpibcast**, **hmpiload**, **hmpirun**,…
docs	This manual
include	Header files
man	Manual pages for HMPI API
lib	Archived HMPI library **libhmpi.a**
tests	Tests for testing HMPI library

6.7 Testing your Installation

After you have successfully installed HMPI, to test the installation, you can test each individual test in the directory "**`$HMPI_HOME/tests`**". Diagnostics are produced showing success or failure of each individual test. Before you test, a virtual parallel machine must be opened.