Wiley Series on Parallel and Distributed Computing
Albert Y. Zomaya, Series Editor

High Performance Heterogeneous Computing

Alexey L. Lastovetsky and Jack J. Dongarra

WILEY

HIGH-PERFORMANCE HETEROGENEOUS COMPUTING

HIGH-PERFORMANCE HETEROGENEOUS COMPUTING

Alexey L. Lastovetsky
University College Dublin

Jack J. Dongarra
University of Tennessee

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey
Published simultaneously in Canada

Library of Congress Cataloging-in-Publication Data

Lastovetsky, Alexey, 1957–
High performance heterogeneous computing / Alexey L. Lastovetsky, Jack Dongarra.
p. cm.—(Wiley series in parallel and distributed computing)
Includes bibliographical references and index.
ISBN 978-0-470-04039-3 (cloth)
1. High performance computing. 2. Heterogeneous computing. 3. Computer networks.
I. Dongarra, J. J. II. Title.
QA76.88.L38 2009
004.6–dc22
2009011754

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

CONTENTS

PREFACE

In recent years, the evolution and growth of the techniques and platforms commonly used for high-performance computing (HPC) in the context of different application domains have been truly astonishing. While parallel computing systems have now achieved certain maturity, thanks to high-level libraries (such as ScaLAPACK, the scalable linear algebra package) or runtime libraries (such as MPI, the message passing interface), recent advances in these technologies pose several challenging research issues. Indeed, current HPC-oriented environments are extremely complex and very difficult to manage, particularly for extreme-scale application problems.

At the very low level, latest-generation CPUs are made of multicore processors that can be general purpose or highly specialized in nature. On the other hand, several processors can be assembled into a so-called symmetrical multiprocessor (SMP), which can also have access to powerful specialized processors, namely graphics processing units (GPUs), which are now increasingly being used for programmable computing resulting from their advent in the video game industry, which significantly reduced their cost and availability. Modern HPC-oriented parallel computers are typically composed of several SMP nodes interconnected by a network. This kind of infrastructure is hierarchical and represents a first class of heterogeneous system in which the communication time between two processing units is different, depending on whether the units are on the same chip, on the same node, or not. Moreover, current hardware trends anticipate a further increase in the number of cores (in a hierarchical way) inside the chip, thus increasing the overall heterogeneity even more toward building extreme-scale systems.

At a higher level, the emergence of heterogeneous computing now allows groups of users to benefit from networks of processors that are already available in their research laboratories. This is a second type of infrastructure where both the network and the processing units are heterogeneous in nature. Specifically, the goal here is to deal with networks that interconnect a large number of heterogeneous computers that can significantly differ from one another in terms of their hardware and software architecture, including different types of CPUs operating at different clock speeds and under different design paradigms, and with different memory sizes, caching strategies, and operating systems.

At the high end, computers are increasingly interconnected together throughout wide area networks to form large-scale distributed systems with high computing capacity. Furthermore, computers located in different laboratories can collaborate in the solution of a common problem. Therefore, the current trends of HPC are clearly oriented toward extreme-scale, complex infrastructures with a great deal of intrinsic heterogeneity and many different hierarchical levels.

It is important to note that all the heterogeneity levels mentioned above are tightly linked. First, some of the nodes in computational distributed environments may be multicore SMP clusters. Second, multicore chips will soon be fully heterogeneous with special-purpose cores (e.g., multimedia, recognition, networking), and not only GPUs, mixed with general-purpose ones. Third, these different levels share many common problems such as efficient programming, scalability, and latency management.

The extreme scale of these environments comes from every level: (a) low level: number of CPUs, number of cores per processor; (b) medium level: number of nodes (e.g., with memory); (c) high level: distributed/large-scale (geographical dispersion, latency, etc.); and (d) application: extreme-scale problem size (e.g., calculation intensive and/or data intensive).

It is realistic to expect that large-scale infrastructures composed of dozens of sites, each composed of several heterogeneous computers, some having thousands of more than 16-core processors, will be available for scientists and engineers. Therefore, the knowledge on how to efficiently use, program, and scale applications on such future infrastructures is very important. While this area is wide open for research and development, it will be unfair to say that it has not been studied yet. In fact, some fundamental models and algorithms for these platforms have been proposed and analyzed. First programming tools and applications have been also designed and implemented. This book gives the state of the art in the field. It analyzes the main challenges of high-performance heterogeneous computing and presents how these challenges have been addressed so far. The ongoing academic research, development, and uses of heterogeneous parallel and distributed computing are placed in the context of scientific computing. While the book is primarily a reference for researchers and developers involved in scientific computing on heterogeneous platforms, it can also serve as a textbook for an advanced university course on high-performance heterogeneous computing.

Alexey L. Lastovetsky
Jack J. Dongarra

ACKNOWLEDGMENTS

We are thankful to Albert Zomaya for his positive attitude to the idea of this book. We would like to thank the anonymous reviewers of the original book proposal for very useful comments and suggestions. We also express our sincere gratitude to the Science Foundation Ireland. Without their support of our collaboration over last five years, this book could not be possible.

PART I

HETEROGENEOUS PLATFORMS: TAXONOMY, TYPICAL USES, AND PROGRAMMING ISSUES

In this part, we outline the existing platforms used for high-performance heterogeneous computing and the typical ways these platforms are used by their end users. We understand a platform as a hardware/software environment used to produce and execute application programs. We also outline programming issues encountered by scientific programmers when they write applications for heterogeneous platforms.

CHAPTER 1

Heterogeneous Platforms and Their Uses

1.1 TAXONOMY OF HETEROGENEOUS PLATFORMS

Heterogeneous platforms used for parallel and distributed computing always include

- multiple processors and
- a communication network interconnecting the processors.

Distributed memory multiprocessor systems can be heterogeneous in many ways. At the same time, there is only one way for such a system to be homogeneous, namely:

- All processors in the system have to be identical and interconnected via a homogeneous communication network, that is, a network providing communication links of the same latency and bandwidth between any pair of processors.
- The same system software (operating system, compilers, libraries, etc.) should be used to generate and execute application programs.

This definition, however, is not complete. One more important restriction has to be satisfied: The system has to be dedicated; that is, at any time it can execute only one application, thus providing all its resources to this application. We will later see how the violation of this restriction can make the system heterogeneous. In practice, the property of dedication can be implemented not only by providing the whole physical system to a single application but also by partitioning the system into logically independent subsystems and providing the nonintersecting partitions to different applications.

Homogeneous distributed memory multiprocessor systems are designed for high-performance parallel computing and are typically used to run a relatively small number of similar parallel applications.

The property of homogeneity is easy to break and may be quite expensive to keep. Any distributed memory multiprocessor system will become heterogeneous if it allows several independent users to simultaneously run their applications on the same set of processors. The point is that, in this case, different identical processors may have different workloads, and hence demonstrate different performances for different runs of the same application depending on external computations and communications.

Clusters of commodity processors are seen as cheap alternatives to very expensive vendor homogeneous distributed memory multiprocessor systems. However, they have many hidden costs required to maintain their homogeneity. First, they cannot be used as multitasking computer systems, allowing several independent users to simultaneously run their applications on the same set of processors. Such a usage immediately makes them heterogeneous because of the dynamic change of the performance of each particular processor. Second, to maintain the homogeneity over time, a full replacement of the system would be required, which can be quite expensive.

Thus, distributed memory multiprocessor systems are naturally heterogeneous, and the property of heterogeneity is an intrinsic property of the overwhelming majority of such systems.

In addition to platforms, which are heterogeneous by nature, one interesting trend is heterogeneous hardware designed by vendors for high-performance computing. The said heterogeneous design is mainly motivated by applications and will be briefly outlined in the next section.

Now we would like to classify the platforms in the increasing order of heterogeneity and complexity and briefly characterize each heterogeneous system. The classes are

- vendor-designed heterogeneous systems,
- heterogeneous clusters,
- local networks of computers (LNCs),
- organizational global networks of computers, and
- general-purpose global networks of computers.

1.2 VENDOR-DESIGNED HETEROGENEOUS SYSTEMS

Heterogeneous computing has seen renewed attention with such examples as the general programming of graphical processing units (GPUs), the Clear Speed (ClearSpeed, 2008, Bristol, UK) Single Instruction Multiple Data (SIMD) attached accelerator, and the IBM (Armonk, NY) Cell architecture (Gschwind *et al.*, 2006).

There has been a marked increase in interest in heterogeneous computing for high performance. Spawned in part by the significant performances demonstrated by special-purpose devices such as GPUs, the idea of finding ways to leverage these industry investments for more general-purpose technical computing has become enticing, with a number of projects mostly in the academia as well as some work in national laboratories. However, the move toward heterogeneous computing is driven by more than the perceived opportunity of "low-hanging fruit." Cray Inc. has described a strategy based on their XT3 system (Vetter *et al.*, 2006), derived from Sandia National Laboratories' Red Storm. Such future systems using an AMD Opteron-based and mesh-interconnected Massively Parallel Processing (MPP) structure will provide the means to support accelerators such as a possible future vector-based processor, or even possibly Field Programmable Gate Arrays (FPGA) devices. The start-up company ClearSpeed has gained much interest in their attached array processor using a custom SIMD processing chip that plugs in to the PCI-X slot of otherwise conventional motherboards. For compute-intensive applications, the possibilities of a one to two order of magnitude performance increase with as little as a 10-W power consumption increase is very attractive.

Perhaps the most exciting advance has been the long-awaited Cell architecture from the partnership of IBM, Sony, and Toshiba (Fig. 1.1). Cell combines the attributes of both multicore and heterogeneous computing. Designed, at least in part, as the breakthrough component to revolutionize the gaming industry in the body of the Sony Playstation 3, both IBM and much of the community look to this part as a major leap in delivered performance. Cell

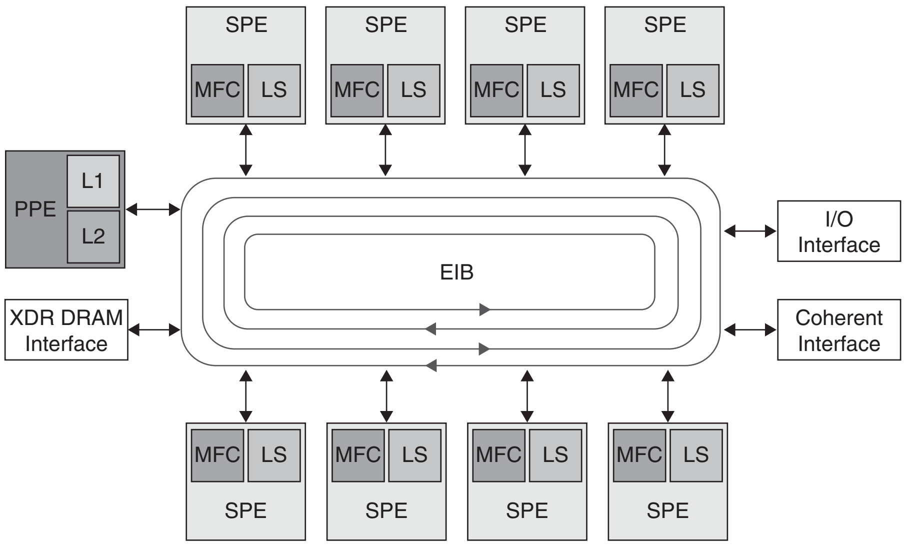

Figure 1.1. The IBM Cell, a heterogeneous multicore processor, incorporates one power processing element (PPE) and eight synergistic processing elements (SPEs). (Figure courtesy of Mercury Computer Systems, Inc.)

incorporates nine cores, one general-purpose PowerPC architecture and eight special-purpose "synergistic processing element (SPE)" processors that emphasize 32-bit arithmetic, with a peak performance of 204 gigaflop/s in 32-bit arithmetic per chip at 3.2 GHz.

Heterogeneous computing, like multicore structures, offer possible new opportunities in performance and power efficiency but impose significant, perhaps even daunting, challenges to application users and software designers. Partitioning the work among parallel processors has proven hard enough, but having to qualify such partitioning by the nature of the work performed and employing multi-instruction set architecture (ISA) environments aggravates the problem substantially. While the promise may be great, so are the problems that have to be resolved. This year has seen initial efforts to address these obstacles and garner the possible performance wins. Teaming between Intel and ClearSpeed is just one example of new and concerted efforts to accomplish this. Recent work at the University of Tennessee applying an iterative refinement technique has demonstrated that 64-bit accuracy can achieve eight times the performance of the normal 64-bit mode of the Cell architecture by exploiting the 32-bit SPEs (Buttari *et al.*, 2007).

Japan has undertaken an ambitious program: the "Kei-soku" project to deploy a 10-petaflops scale system for initial operation by 2011. While the planning for this initiative is still ongoing and the exact structure of the system is under study, key activities are being pursued with a new national High Performance Computing (HPC) Institute being established at RIKEN (2008). Technology elements being studied include various aspects of interconnect technologies, both wire and optical, as well as low-power device technologies, some of which are targeted to a 0.045-μm feature size. NEC, Fujitsu, and Hitachi are providing strong industrial support with academic partners, including University of Tokyo, Tokyo Institute of Technology, University of Tsukuba, and Keio University among others. The actual design is far from certain, but there are some indications that a heterogeneous system structure is receiving strong consideration, integrating both scalar and vector processing components, possibly with the addition of special-purpose accelerators such as the MD-Grape (Fukushige *et al.*, 1996). With a possible budget equivalent to over US$1 billion (just under 1 billion euros) and a power consumption of 36 MW (including cooling), this would be the most ambitious computing project yet pursued by the Asian community, and it is providing strong leadership toward inaugurating the Petaflops Age (1–1000 petaflops).

1.3 HETEROGENEOUS CLUSTERS

A heterogeneous cluster (Fig. 1.2) is a dedicated system designed mainly for high-performance parallel computing, which is obtained from the classical homogeneous cluster architecture by relaxing one of its three key properties, thus leading to the situation wherein:

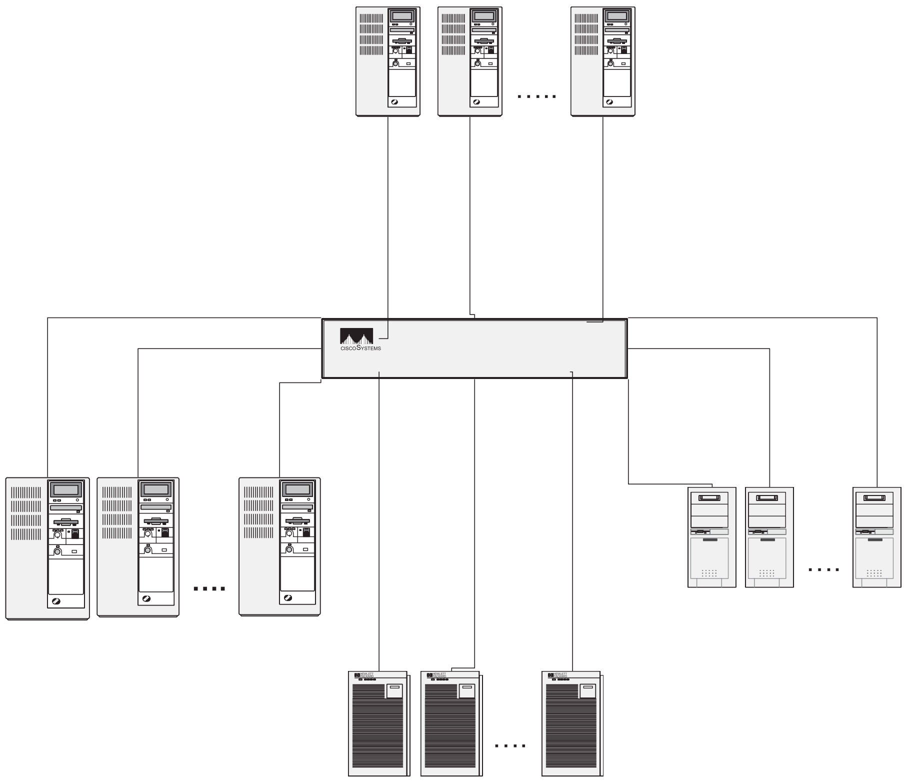

Figure 1.2. A heterogeneous switch-enabled computational cluster with processors of different architectures.

- Processors in the cluster may not be identical.
- The communication network may have a regular but heterogeneous structure. For example, it can consist of a number of faster communication segments interconnected by relatively slow links. Such a structure can be obtained by connecting several homogeneous clusters in a single multicluster.
- The cluster may be a multitasking computer system, allowing several independent users to simultaneously run their applications on the same set of processors (but still dedicated to high-performance parallel computing). As we have discussed, this, in particular, makes the performance characteristics of the processors dynamic and nonidentical.

The heterogeneity of the processors can take different forms. The processors can be of different architectures. They may be of the same architecture but of different models. They may be of the same architecture and model but running different operating systems. They may be of the same architecture and model and running the same operating system but configured differently or using different basic softwares to produce executables (compilers, runtime libraries, etc.). All the differences in the systems' hardware and software can have an impact on the performance and other characteristics of the processors.

In terms of parallel programming, the most demanding is a multitasking heterogeneous cluster made up of processors of different architectures interconnected via a heterogeneous communication network.

1.4 LOCAL NETWORK OF COMPUTERS (LNC)

In the general case, an LNC consists of diverse computers interconnected via mixed network equipment (Fig. 1.3). By its nature, LNCs are multiuser and multitasking computer systems. Therefore, just like highly heterogeneous clusters, LNCs consist of processors of different architectures, which can dynamically change their performance characteristics, interconnected via a heterogeneous communication network.

Unlike heterogeneous clusters, which are parallel architectures designed mainly for high-performance computing, LNCs are general-purpose computer systems typically associated with individual organizations. This affects the heterogeneity of this platform in several ways. First, the communication network of a typical LNC is not regular and balanced as in heterogeneous clusters. The topology and structure of the communication network in such an LNC are determined by many different factors, among which high-performance computing is far from being a primary one if considered at

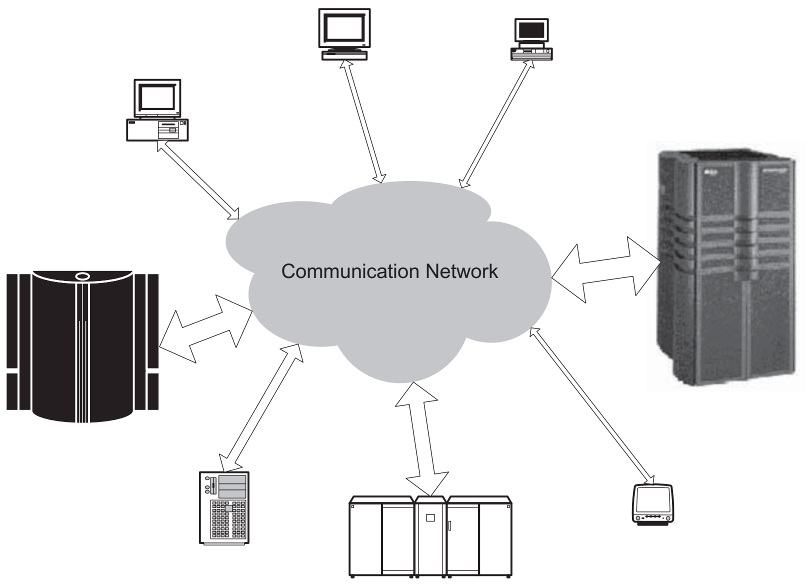

Figure 1.3. A local network of computers.

all. The primary factors include the structure of the organization, the tasks that are solved on the computers of the LNC, the security requirements, the construction restrictions, the budget limitations, and the qualification of technical personnel, etc. An additional important factor is that the communication network is constantly being developed rather than fixed once and for all. The development is normally occasional and incremental; therefore, the structure of the communication network reflects the evolution of the organization rather than its current snapshot. All the factors make the communication network of the LNC extremely heterogeneous and irregular. Some communication links in this network may be of very low latency and/or low bandwidth.

Second, different computers may have different functions in the LNC. Some computers can be relatively isolated. Some computers may provide services to other computers of the LNC. Some computers provide services to both local and external computers. These result to different computers having different levels of integration into the network. The heavier the integration, the more dynamic and stochastic the workload of the computer is, and the less predictable its performance characteristics are. Another aspect of this functional heterogeneity is that a heavy server is normally configured differently compared with ordinary computers. In particular, a server is typically configured to avoid paging, and hence to avoid any dramatic drop in performance with the growth of requests to be served. At the same time, this results in the abnormal termination of any application that tries to allocate more memory than what fits into the main memory of the computer, leading to the loss of continuity of its characteristics.

Third, in general-purpose LNCs, different components are not as strongly integrated and controlled as in heterogeneous clusters. LNCs are much less centralized computer systems than heterogeneous clusters. They consist of relatively autonomous computers, each of which may be used and administered independently by its users. As a result, their configuration is much more dynamic than that of heterogeneous clusters. Computers in the LNC can come and go just because their users switch them on and off or reboot them.

1.5 GLOBAL NETWORK OF COMPUTERS (GNC)

Unlike an LNC, all components of which are situated locally, a GNC includes computers that are geographically distributed (Fig. 1.4). There are three main types of GNCs, which we briefly present in the increasing order of their heterogeneity.

The first type of GNC is a dedicated system for high-performance computing that consists of several interconnected homogeneous distributed memory multiprocessor systems or/and heterogeneous clusters. Apart from the geographical distribution of its components, such a computer system is similar to heterogeneous clusters.

The second type of GNC is an organizational network. Such a network comprises geographically distributed computer resources of some individual

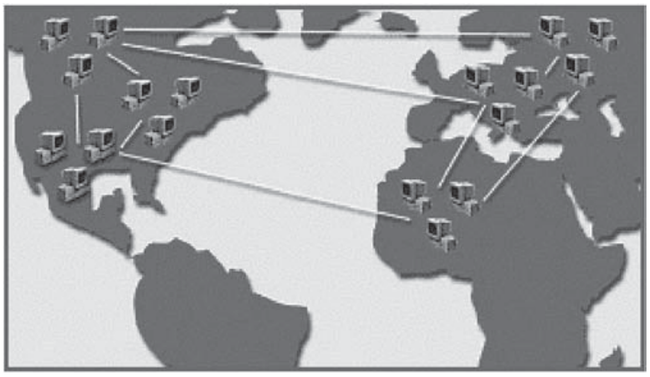

Figure 1.4. A global network of computers.

organization. The organizational network can be seen as a geographically extended LNC. It is typically managed by a strong team of hardware and software experts. Its levels of integration, centralization, and uniformity are often even higher than that of LNCs. Therefore, apart from the geographical distribution of their components, organizational networks of computers are quite similar to LNCs.

Finally, the third type of GNC is a general-purpose GNC. Such a network consists of individual computers interconnected via the Internet. Each of the computers is managed independently. This is the most heterogeneous, irregular, loosely integrated, and dynamic type of heterogeneous network.

1.6 GRID-BASED SYSTEMS

Grid computing received a lot of attention and funding over the last decade, while the concepts and ideas have been around for a while (Smarr and Catlett, 1992). The definitions of Grid computing are various and rather vague (Foster, 2002; GridToday, 2004). Grid computing is declared as a new computing model aimed at the better use of many separate computers connected by a network. Thus, the platform targeted by Grid computing is a heterogeneous network of computers. Therefore, it is important to formulate our vision of Grid-based heterogeneous platforms and their relation to traditional distributed heterogeneous platforms in the context of scientific computing on such platforms.

As they are now, Grid-based systems provide a mechanism for a single log-in to a group of resources. In Grid-based systems, the user does not need to separately log in at each session to each of the resources that the user wants to access. The Grid middleware will do it for the user. It will keep a list of available resources that the user have discovered and add them to a list in the past. Upon the user's log-in to the Grid-based system, it will detect which of the resources are available now, and it will log in to all the available resources on behalf of the user. This is the main difference of Grid-based systems from traditional distributed systems, where individual access to the distributed resources is the full responsibility of the user.

A number of services can be build on top of this mechanism, thus forming a Grid operating environment. There are different models of the operating environment supported by different Grid middlewares such as Globus (2008) and Unicore (2008). From the scientific computing point of view, it is important to note that as soon as the user has logged in to all distributed resources, then there is no difference between a traditional heterogeneous distributed system and a Grid-based heterogeneous distributed system.

1.7 OTHER HETEROGENEOUS PLATFORMS

Of course, our list of heterogeneous distributed memory multiprocessor systems is not comprehensive. We only outlined systems that are most relevant for scientific computing. Some other examples of heterogeneous sets of interconnected processing devices are

- mobile telecommunication systems with different types of processors, from ones embedded into mobile phones to central computers processing calls, and
- embedded control multiprocessor systems (cars, airplanes, spaceships, household, etc.).

1.8 TYPICAL USES OF HETEROGENEOUS PLATFORMS

In this section, we outline how heterogeneous networks of computers are typically used by their end users. In general, heterogeneous networks are used traditionally, for parallel computing, or for distributed computing.

1.8.1 Traditional Use

The traditional use means that the network of computers is used just as an extension of the user's computer. This computer can be serial or parallel. The application to be run is a traditional application, that is, one that can be executed on the user's computer. The code of the application and input data are provided by the user. The only difference from the fully traditional execution of the application is that it can be executed not only on the user's computer but also on any other relevant computer of the network. The decision where to execute one or other applications is made by the operating environment and is mainly aimed at the better utilization of available computing resources (e.g., at higher throughput of the network of computers as a whole multiuser computer system). Faster execution of each individual application is not the main goal of the operating environment, but it can be achieved for some applications as a side effect of its scheduling policy. This use of the heterogeneous network assumes that the application, and hence the software, is portable and can be run on another computing resource. This assumption may not be true for some applications.

1.8.2 Parallel Computing

A heterogeneous network of computers can be used for parallel computing. The network is used as a parallel computer system in order to accelerate the solution of a single problem. In this case, the user provides a dedicated parallel application written to efficiently solve the problem on the heterogeneous network of computers. High performance is the main goal of this type of use. As in the case of traditional use, the user provides both the (source) code of the application and input data. In the general case, when all computers of the network are of a different architecture, the source code is sent to the computers, where it is locally compiled. All the computers are supposed to provide all libraries necessary to produce local executables.

1.8.3 Distributed Computing

A heterogeneous network of computers can be also used for distributed computing. In the case of parallel computing, the application can be executed on the user's computer or on any other single computer of the network. The only reason to involve more than one computer is to accelerate the execution of the application. Unlike parallel computing, distributed computing deals with situations wherein the application cannot be executed on the user's computer because not all components of the application are available on this computer. One such situation is when some components of the code of the application cannot be provided by the user and are only available on remote computers. There are various reasons behind this: the user's computer may not have the resources to execute such a code component; the efforts and amount of resources needed to install the code component on the user's computer are too significant compared with the frequency of its execution; this code may be not available for installation; or it may make sense to execute this code only on the remote processor (say, associated with an ATM machine), etc.

Another situation is when some components of input data for this application cannot be provided by the user and reside on remote storage devices. For example, the size of the data may be too big for the disk storage of the user's computer, the data for the application are provided by some external party (remote scientific device, remote data base, remote application, and so on), or the executable file may not be compatible with the machine architecture.

The most complex is the situation when both some components of the code of the application and some components of its input data are not available on the user's computer.

CHAPTER 2

Programming Issues

Programming for heterogeneous networks of computers is a difficult task. Among others, performance, fault tolerance, and arithmetic heterogeneity are perhaps the most important and challenging issues of heterogeneous parallel and distributed programming.

Performance is one of the primary issues of parallel programming for any parallel architecture, but it becomes particularly challenging for programming for parallel heterogeneous networks. Performance is also one of the primary issues of high-performance distributed computing.

Fault tolerance has always been one of the primary issues of distributed computing. Interestingly, this has not been the case for parallel applications running on traditional homogeneous parallel architectures. The probability of unexpected resource failures in a centralized dedicated parallel computer system was quite small because the system had a relatively small number of processors. This only becomes an issue for modern large-scale parallel systems counting tens of thousands of processors with different interconnection schemes. At the same time, this probability reaches quite high figures for common networks of computers of even a relatively small size. First, any individual computer in such a network may be switched off or rebooted unexpectedly for other users in the network. The same may happen with any other resource in the network. Second, not all elements of the common network of computers are equally reliable. These factors make fault tolerance a desirable feature for parallel applications intended to run on common networks of computers; and the longer the execution time of the application is, the more critical the feature becomes.

Arithmetic heterogeneity has never been an issue of parallel programming for traditional homogeneous parallel architectures. All arithmetic data types are uniformly represented in all processors of such a system, and their transfer between the processors does not change their value. In heterogeneous platforms, the same arithmetic data type may have different representations in different processors. In addition, arithmetic values may change in the heterogeneous communication network during transfer even between processors with the same data representation. Thus, arithmetic heterogeneity is a new parallel programming issue specific to heterogeneous parallel computing. The finer the granularity of the parallel application is and the more communications its execution involves, the more frequently arithmetic values from different processors are mixed in computations, and hence the more serious this issue becomes. At the same time, if the problem and the method of solution is not ill conditioned, then arithmetic heterogeneity is not a serious issue for distributed computing.

In this chapter, we analyze these three issues with respect to parallel and distributed programming for heterogeneous networks of computers.

2.1 PERFORMANCE

In this section, we outline the performance issues of scientific programming for heterogeneous platforms and discuss how different aspects of heterogeneity contribute their specific challenges to the problem of achieving top performance on such platforms. We start with the very basic implications from the heterogeneity of processors. Then, we analyze how memory heterogeneity, memory constraints, heterogeneity of integration of the processors into the network, and unbalance between the performance of the processors and the performance of the communication network further complicate the performance issue. Finally, we look at the performance-related challenges posed by the heterogeneity of communication networks.

An immediate implication from the heterogeneity of processors in a network of computers is that the processors run at different speeds. A good parallel application for a homogeneous distributed memory multiprocessor system tries to evenly distribute computations over available processors. This very distribution ensures the maximal speedup on the system consisting of identical processors. If the processors run at different speeds, faster processors will quickly perform their part of the computations and begin waiting for slower processors at points of synchronization and data transfer. Therefore, the total time of computations will be determined by the time elapsed on the slowest processor. In other words, when executing parallel applications, which evenly distribute computations among available processors, a set of heterogeneous processors will demonstrate the same performance as a set of identical processors equivalent to the slowest processor in the heterogeneous set.

Therefore, a good parallel application for the heterogeneous platform must distribute computations unevenly taking into account the difference in processor speed. The faster the processor is, the more computations it must perform. Ideally, in the case of independent parallel computations (that is, computations on parallel processors without synchronization or data transfer), the volume of computations performed by a processor should be proportional to its speed.

Distribution of computations over the processors in proportion to their speed assumes that the programmers know at least the relative speeds of the processor in the form of positive constants. The performance of the corresponding application will strongly depend on the accuracy of estimation of the relative speed. If this estimation is not accurate enough, the load of the processors will be unbalanced, resulting in poorer execution performance. Unfortunately, the problem of accurate estimation of the relative speed of processors is not as easy as it may look. Of course, if we consider two processors, which only differ in clock rate, it is not a problem to accurately estimate their relative speed. We can use a single test code to measure their relative speed, and the relative speed will be the same for any application. This approach may also work if the processors used in computations have very similar architectural characteristics.

However, if we consider processors of very different architectures, the situation changes drastically. Everything in the processors may be different: the set of instructions, the number of instruction execution units, the number of registers, the structure of memory hierarchy, the size of each memory level, and so on. Therefore, the processors may demonstrate different relative speeds for different applications. Moreover, processors of the same architecture but of different models or configurations may also demonstrate different relative speeds on different applications. Even different applications of the same narrow class may be executed by two different processors at significantly different relative speeds.

Thus, the relative speeds of heterogeneous processors are application specific, which makes the problem of their accurate estimation nontrivial. The test code used to measure the relative speed should be carefully designed for each particular application.

Another complication of the problem comes up if the heterogeneous platform allows for multitasking, wherein several independent users can simultaneously run their applications on the same set of processors. In this case, the relative speed of the processors can dynamically change depending on the external load.

The accuracy of estimation of the relative speed of the processors not only depends on how representative is the test code used to obtain the relative speed or how frequently this estimation is performed during the execution of the application. Some objective factors do not allow us to estimate the speed of some processors accurately enough. One of these factors is the level of integration of the processor into the network. As we have discussed in Chapter 1, in general-purpose local and global networks integrated into the Internet, most computers and their operating systems periodically run some routine processes interacting with the Internet, and some computers act as servers for other computers. This results in constant unpredictable fluctuations in the workload of processors in such a network. This changing transient load will cause fluctuations in the speed of processors, in that the speed of the processor will vary when measured at different times while executing the same task. We would like to stress that this additional challenge is specific to general-purpose local and global heterogeneous networks. Heterogeneous clusters dedicated to high-performance computing are much more regular and predictable in this respect.

So far, we implicitly assumed that the relative speed of processors being application specific does not depend on the size of the computational task solved by the processors. This assumption is quite realistic if the code executed by the processors fully fits into the main memory. However, as soon as the restriction is relaxed, it may not be realistic anymore. The point is that beginning from some problem size, a task of the same size will still fit into the main memory of some processors and will 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 nonpaging processors as soon as the problem size exceeds the critical value. Moreover, even if two processors of different architectures have almost the same size of main memory, they may employ different paging algorithms, resulting in different levels of speed degradation for a task of the same size, which again leads to the change of their relative speed as the problem size exceeds the threshold causing the paging. Thus, memory heterogeneity and paging effects significantly complicate the problem of accurate estimation of the relative speed of heterogeneous processors. Estimations obtained in the absence of paging may be inaccurate when the paging occurs and vice versa.

Yet another additional challenge is also related to memory and specific to general-purpose networks of computers. It occurs when the network includes computers that are configured to avoid paging. This is typical of computers used as a main server. If the computational task allocated to such a computer does not fit into the main memory, it will crash. In this case, the problem of optimal distribution of computations over the processors of the network becomes more difficult, having the additional restriction on the maximal size of tasks to be assigned to some processors.

One more factor that has a significant impact on the optimal distribution of computations over heterogeneous processors has not been taken into account so far. This factor is the communication network interconnecting the processors, even if the network is homogeneous. This factor can only be neglected if the contribution of communication operations in the total execution time of the application is negligibly small compared with that of computations. Communication networks in heterogeneous platforms are typically not as well balanced with the number and speed of the processors as those in dedicated homogeneous high-performance multiprocessor systems. Therefore, it is much more likely that the cost of communication for some applications will not compensate the gains due to parallelization if all available processors are involved in its execution. In this case, the problem of optimal distribution of computations over the processors becomes much more complex as the space of possible solutions will significantly increase, including distributions not only over all available processors but also over subsets of processors.

For distributed memory platforms with homogeneous communication networks providing parallel communication links of the same performance between each pair of processors, the problem of minimizing the communication cost of the application can typically be reduced to the problem of minimizing the total volume of communications. The heterogeneity of the communication network changes the situation, making the problem of minimizing the communication cost much more difficult. Indeed, a larger amount of data communicated through faster links only may lead to less overall communication cost than a smaller amount of data communicated through all the links, both fast and slow. Even if each communication link in such a heterogeneous platform is characterized just by one number, the corresponding optimization problem will have to deal with up to p^2 additional parameters, where p is the number of processors.

The heterogeneity of the communication network also makes the optimal distribution of computations, minimizing the overall computation/communication cost, much more of a challenging task. For example, even in the case of homogeneous processors interconnected by a heterogeneous network, such an optimal distribution can be uneven. Additional challenges are brought by possible dynamic changes of the performance characteristics of the communication links due to multitasking or integration into the Internet.

2.2 FAULT TOLERANCE

In this section, we outline the fault tolerance issues of scientific programming for heterogeneous platforms and discuss how different aspects of heterogeneity add their specific challenges to the problem of tolerating failures on such platforms. The ideas that follow in this section can be applied to both heterogeneous and homogeneous processing.

The unquenchable desire of scientists to run ever larger simulations and analyze ever larger data sets is fueling a relentless escalation in the size of supercomputing clusters from hundreds to thousands, to even tens of thousands of processors. Unfortunately, the struggle to design systems that can scale up in this way also exposes the current limits of our understanding of how to efficiently translate such increases in computing resources into corresponding increases in scientific productivity. One increasingly urgent part of this knowledge gap lies in the critical area of *reliability and fault tolerance*.

Even when making generous assumptions on the reliability of a single processor, it is clear that as the processor count in high-end clusters and heterogeneous systems grows into the tens of thousands, the mean time to failure (MTTF) will drop from hundreds of days to a few hours, or less. The type of 100,000-processor machines projected in the next few years can expect to experience processor failure almost daily, perhaps hourly. Although today's architectures are robust enough to incur process failures without suffering complete system failure, at this scale and failure rate, the only technique available to application developers for providing fault tolerance within the current parallel programming model—checkpoint/restart—has performance and conceptual limitations that make it inadequate for the future needs of the communities that will use these systems.

After a brief decline in popularity, distributed memory machines containing large numbers of processors have returned to fulfill the promise of delivering high performance to scientific applications. While it would be most convenient for application scientists to simply port their message passing interface (MPI) codes to these machines, perhaps instrument them with a global checkpointing system, and then sit back and enjoy the performance improvements, there are several features of these machines and their typical operating environments that render this impossible:

- *Large Numbers of Processors Mean More Failures.* Builders of distributed machines are targeting them to have tens, or even hundreds, of thousands of processors (e.g., the Blue Gene [IBM] has 128,000 processors). While that represents a great potential of computing power, it also represents a great potential increase in the system failure rate. Given independent failures, if the failure rate of one processor is X, then the rate of failure of the first processor in an N processor system is NX. Thus, if the single processor rate of failure is one per year, the rate of processor failure in a system of 128,000 processors is one per 6 hours! Clearly, failures must be accounted for in the programming system.
- *Message-Passing Systems Must Tolerate Single-Processor Failures.* As a by-product of the previous point, the programming environment of such systems must be able to identify and tolerate single-processor failures. Historically, MPI systems crash upon processor failures, requiring applications to utilize global checkpointing and restart to tolerate them. However, such high failure rates imply that global checkpointing approaches are too inefficient.
- *Limited Bandwidth to Shared, Stable Storage.* High-performance machines pay a great deal of attention to providing high-performance storage capabilities. However, with so many processors and hierarchies of networks, access to shared storage will necessarily be a bottleneck. Although at peak input/output (I/O) performance the needs of global checkpointing may be supported, such checkpointing will seriously conflict with both messaging and regular I/O of the application program.

Fault tolerance techniques can usually be divided into three big branches and some hybrid techniques. The first branch is *messaging logging*. In this branch, there are three subbranches: *pessimistic messaging logging*, *optimistic messaging logging*, and *casual messaging logging*. The second branch is *checkpointing and rollback recovery*. There are also three subbranches in this branch: *network disk-based checkpointing and rollback recovery*, *diskless checkpointing and rollback recovery*, and *local disk-based checkpointing and rollback recovery*. The third branch is *algorithm-based fault tolerance*.

There has been much work on fault tolerance techniques for high-performance computing. These efforts come in basically four categories and can be adapted to heterogeneous computing.

1. *System-Level Checkpoint/Message Logging:* Most fault tolerance schemes in the literature belong to this category. The idea of this approach is to incorporate fault tolerance into the system level so that the application can be recovered automatically without any efforts from the application programmer. The most important advantage of this approach is its transparency. However, due to lack of knowledge about the semantics of the application, the system typically backs up all the processes and logs all messages, thus often introducing a huge amount of fault tolerance overhead.
2. *Compiler-Based Fault Tolerance Approach:* The idea of this approach is to exploit the knowledge of the compiler to insert the checkpoint at the best place and to exclude irrelevant memory areas to reduce the size of the checkpoint. This approach is also transparent. However, due to the inability of the compiler to determine the state of the communication channels at the time of the checkpoint, this approach is difficult to use in parallel/distributed applications that communicate through message passing.
3. *User-Level Checkpoint Libraries:* The idea of this approach is to provide some checkpoint libraries to the programmer and let the programmer decide where, when, and what to checkpoint. The disadvantage of this approach is its nontransparency. However, due to the involvement of the programmer in the checkpoint, the size of the checkpoint can be reduced considerably, and hence the fault tolerance overhead can also be reduced considerably.
4. *Algorithmic Fault Tolerance Approach:* The idea of this approach is to leverage the knowledge of algorithms to reduce the fault tolerance overhead to the minimum. In this approach, the programmer has to decide not only where, when, and what to checkpoint but also how to do the checkpoint, and hence the programmer must have deep knowledge about the application. However, if this approach can be incorporated into widely used application libraries such as ScaLAPACK and PETSc, then it is possible to reduce both the involvement of the application programmer and the overhead of the fault tolerance to a minimum.

2.3 ARITHMETIC HETEROGENEITY

There are special challenges associated with writing reliable numerical software on systems containing heterogeneous platforms, that is, processors that may do floating-point arithmetic differently. This includes not just machines with completely different floating-point formats and semantics, such as Cray vector computers running *Cray arithmetic* versus workstations running IEEE-standard floating-point arithmetic, but even supposedly identical machines running with different compilers, or even just different compiler options or runtime environments.

The basic problem occurs when making *data dependent branches* on different platforms. The flow of an algorithm is usually data dependent, and therefore slight variations in the data may lead to different processors executing completely different sections of code.

Now we attempt a definition of an arithmetically heterogeneous platform. The three main issues determining the classification are the hardware, the communication layer, and the software (operating system, compiler, compiler options). Any differences in these areas can potentially affect the behavior of the application. Specifically, the following conditions must be satisfied before a platform can be considered *arithmetically homogeneous*:

1. The hardware of each processor guarantees the same storage representation and the same results for operations on floating-point numbers.
2. If a floating-point number is communicated between processors, the communication layer guarantees the exact transmittal of the floating-point value.
3. The software (operating system, compiler, compiler options) on each processor also guarantees the same storage representation and the same results for operations on floating-point numbers.

We regard an *arithmetically homogeneous machine* as one, which satisfies condition 1. An *arithmetically homogeneous network* is a collection of homogeneous machines, which additionally satisfies condition 2. Finally, an *arithmetically homogeneous platform* is a homogeneous network, which satisfies condition 3. We can then make the obvious definition that an *arithmetically heterogeneous platform* is one that is not homogeneous. The requirements for an arithmetically homogeneous platform are quite stringent and are frequently not met in networks of workstations, or in PCs, even when each computer in the network is the same model.

Some areas of distinction are obvious, such as a difference in the architecture of two machines or the type of communication layer implemented. Some hardware and software issues, however, can potentially affect the behavior of the application and be difficult to diagnose. For example, the determination of machine parameters such as machine precision, overflow, and underflow, the implementation of complex arithmetic such as complex division, or the handling of NaNs and subnormal numbers could differ. Some of these subtleties may only become apparent when the arithmetic operations occur on the edge of the range of representable numbers.

The difficult question that remains unanswered for scientific programmers is: When can we *guarantee* that heterogeneous computing is safe? There is also the question of just how much additional programming effort should we expend to gain additional robustness.

Machine parameters such as the relative machine precision, the underflow and overflow thresholds, and the smallest value, which can be safely reciprocated, are frequently used in numerical linear algebra computations, as well as in many other numerical computations. Without due care, variations in these values between processors can cause problems, such as those mentioned above. Many such problems can be eliminated by using the *largest* machine precision among all participating processors.

The IEEE standard for binary floating-point arithmetic (IEEE, 1985) specifies how machines conforming to the standard should represent floating-point values. We refer to machines conforming to this standard as *IEEE machines*.[1] Thus, when we communicate floating-point numbers between IEEE machines, we might hope that each processor has the same value. This is a reasonable hope and will often be realized. For example, external data representation (XDR) (SunSoft, 1993), uses the IEEE representation for floating-point numbers, and therefore a message-passing system that uses XDR will communicate floating-point numbers without change.[2] Parallel Virtual Machine (PVM) is an example of a system that uses XDR. MPI suggests the use of XDR but does not mandate its use (Snir *et al.*, 1996). Unless we have additional information about the implementation, we cannot assume that floating-point numbers will be communicated without change on IEEE machines when using MPI. Note that there is also an IEEE standard concerned with standardizing data formats to aid data conversion between processors (IEEE, 1994).

Rigorous testing of the ScaLAPACK package, particularly for floating-point values close to the edge of representable numbers, exposed additional dangers that must be avoided in floating-point arithmetic (Demmel *et al.*, 2007). For example, it is a sad reflection that some compilers still do not implement complex arithmetic carefully. In particular, unscaled complex division still occurs on certain architectures, leading to unnecessary overflow.[3] To handle this difficulty, ScaLAPACK, as LAPACK, restricts the range of representable numbers by a call to routine PDLABAD (in double precision), the equivalent of the LAPACK routine DLABAD, which replaces the smallest and largest representable numbers by their respective square roots in order to give protection from underflow or overflow on machines that do not take the care to scale on operations such as complex division. PDLABAD calls DLABAD locally on each process and then communicates the minimum and maximum values, respectively. Arguably, there should be separate routines for real and complex arithmetic, but there is a hope that the need for DLABAD will eventually disappear.

This is particularly irritating if one machine in a network is causing us to impose unnecessary restrictions on all the machines in the network, but without such a restriction, catastrophic results can occur during computations near the overflow or underflow thresholds.

Another problem encountered during the testing is in the way that subnormal (denormalized) numbers are handled on certain (near) IEEE

[1] It should be noted that there is also a radix independent standard (IEEE, 1987).

[2] It is not clear whether or not this can be assumed for subnormal (denormalized) numbers.

[3] At the time of testing ScaLAPACK version 1.2, the HP9000 exhibited this behavior.

architectures. By default, some architectures flush subnormal numbers to zero.[4] Thus, if the computation involves numbers near underflow and a subnormal number is communicated to such a machine, the computational results may be invalid and the subsequent behavior unpredictable. Often such machines have a compiler switch to allow the handling of subnormal numbers, but it can be nonobvious and we cannot guarantee that users will use such a switch.

This behavior occurred during the heterogeneous testing of the linear least squares routines when the input test matrix was a full-rank matrix scaled near underflow. During the course of the computation, a subnormal number was communicated, then this value was unrecognized on receipt, and a floating-point exception was flagged. The execution on the processor was killed, subsequently causing the execution on the other processors to hang. A solution would be to replace subnormal numbers either with zero, or with the nearest normal number, but we are somewhat reluctant to implement this solution as ScaLAPACK does not seem to be the correct software level at which to address the problem.

The suggestions made so far certainly do not solve all of the problems. We are still left with major concerns for problems associated with varying floating-point representations and arithmetic operations between different processors, different compilers, and different compiler options.

We tried to illustrate some of the potential difficulties concerned with floating-point computations on heterogeneous platforms. Some of these difficulties are straightforward to address, while others require considerably more thought. All of them require some additional level of defensive programming to ensure the usual standards of reliability that users have come to expect from packages such as LAPACK and ScaLAPACK.

We have presented reasonably straightforward solutions to the problems associated with floating-point machine parameters and global values, and we have discussed the use of a controlling process to solve some of the difficulties of algorithmic integrity. This can probably be used to solve most of these problems. Although in some cases, this might be at the expense of considerable additional overhead, usually in terms of additional communication, which is also imposed on an arithmetically homogeneous network unless we have separate code for the homogeneous case. Unless we can devise a satisfactory test for arithmetic homogeneity, and hence have separate paths within the code, a separate code would defeat the aim of portability.

A topic that we have not discussed is that of the additional testing necessary to give confidence in heterogeneous platforms. The testing strategies that are needed are similar to those already employed in reputable software packages such as LAPACK, but it may be very hard to produce actual test examples that would detect incorrect implementations of the algorithms because, as we have seen, the failures are likely to be very sensitive to the computing environment and, in addition, may be nondeterministic.

[4] The DEC Alpha, at the time of writing, is an example.

PART II

PERFORMANCE MODELS OF HETEROGENEOUS PLATFORMS AND DESIGN OF HETEROGENEOUS ALGORITHMS

In this part, we present the state of the art in two related fields—modeling the performance of heterogeneous platforms for high-performance computing and design and analysis of heterogeneous algorithms with the models.

CHAPTER 3

Distribution of Computations with Constant Performance Models of Heterogeneous Processors

3.1 SIMPLEST CONSTANT PERFORMANCE MODEL OF HETEROGENEOUS PROCESSORS AND OPTIMAL DISTRIBUTION OF INDEPENDENT UNITS OF COMPUTATION WITH THIS MODEL

Heterogeneity of processors is one of the main sources of performance programming issues. As we have seen in Chapter 2, the immediate and most important performance-related implication from the heterogeneity of processors is that the processors run at different speeds. The simplest performance model, capturing this feature and abstracting from the others, sees a heterogeneous network of computers as a set of interconnected processors, each of which is characterized by a single positive constant representing its speed. Two important parameters of the model include

- p, the number of the processors, and
- $S = \{s_1, s_2, \ldots, s_p\}$, the speeds of the processors.

The speed of the processors can be either *absolute* or *relative*. The absolute speed of the processors is understood as the number of computational units performed by the processor per one time unit. The relative speed of the processor can be obtained by the normalization of its absolute speed so that $\Sigma_{i=1}^{p} s_i = 1$. Some researchers also use the reciprocal of the speed, which they call the execution time of the processor. For example, if s_i is the absolute speed of processor P_i, then $t_i = \frac{1}{s_i}$ will be the execution time of this processor giving the number of time units needed to perform one unit of computation on processor P_i.

High-Performance Heterogeneous Computing, by Alexey L. Lastovetsky and Jack J. Dongarra

The performance model presented above does not have parameters describing the communication network. Nonetheless, as we will later see, even in the framework of such a simple model, the communication cost of parallel algorithms can be taken into account.

Now we consider a simple but fundamental optimization problem with this model—the problem of optimal distribution of independent equal units of computation over a set of heterogeneous processors. The solution of this problem is used as a basic building block in solutions of more complicated optimization problems.

The problem can be formulated as follows. Given n independent units of computations, each of equal size (i.e., each requiring the same amount of work), how can we assign these units to p ($p < n$) physical processors $P_1, P_2, \ldots, P_p$ of respective speeds $s_1, s_2, \ldots, s_p$ so that the workload is best balanced? Here, the speed s_i of processor P_i is understood as the number of units of computation performed by processor P_i per one time unit.

Then, how do we distribute the computational units to processors? The intuition says that the load of P_i should be proportional to s_i. As the loads (i.e., the numbers of units of computation) on each processor must be integers, we use the following two-step algorithm to solve the problem. Let n_i denote the number of units of computation allocated to processor P_i. Then, the overall execution time obtained with allocation $(n_1, n_2, \ldots, n_p)$ is given by $\max_i \frac{n_i}{s_i}$. The optimal solution will minimize the overall execution time (without taking into account communication).

Algorithm 3.1 (Beaumont *et al.*, 2001a). Optimal distribution for n independent units of computation over p processors of speeds $s_1, s_2, \ldots, s_p$:

- **Step 1: Initialization.** Approximate the n_i so that $\frac{n_i}{s_i} \approx const$ and $n_1 + n_2 + \ldots + n_p \le n$. Namely, we let $n_i = \left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor$ for $1 \le i \le p$.
- **Step 2: Refining.** Iteratively increment some n_i until $n_1 + n_2 + \ldots + n_p = n$ as follows:

```
while ($n_1 + n_2 + \ldots + n_p < n$) {
 find $k \in \{1, \ldots, p\}$ such that $\frac{n_k + 1}{s_k} = \min_{i=1}^{p} \frac{n_i + 1}{s_i}$;
 $n_k = n_k + 1$;
}
```

Proposition 3.1 (Beaumont *et al.*, 2001a). Algorithm 3.1 gives the optimal distribution.

See Appendix A for proof.

Proposition 3.2. The complexity of Algorithm 3.1 is $O(p^2)$.

Proof. The complexity of the initialization step is $O(p)$. The complexity of one iteration of the refining is $O(p)$. After the initialization step, $n_1 + n_2 + \ldots + n_p \geq n - p$. Therefore, there will be at most p iterations of the refining. Hence, the overall complexity of the algorithm will be $O(p^2)$. *End of proof.*

Proposition 3.3 (Beaumont *et al.*, 2001a). The complexity of Algorithm 3.1 can be reduced down to $O(p \times \log p)$ using *ad hoc* data structures.

The algorithm is widely used as a basic building block in the design of many heterogeneous parallel and distributed algorithms. One simple example is the following parallel algorithm of multiplication of two dense square $n \times n$ matrices, $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$, on p heterogeneous processors:

- First, we partition matrices ***A*** and ***C*** identically into p horizontal slices such that there will be one-to-one mapping between these slices and the processors. Each processor will store its slices of matrices ***A*** and ***C*** and the whole matrix ***B*** as shown in Figure 3.1 for $p = 3$.
- All processors compute their ***C*** slices in parallel such that each element c_{ij} in ***C*** is computed as $c_{ij} = \sum_{k=0}^{n-1} a_{ik} \times b_{kj}$.

The key step of this algorithm is the partitioning of matrices ***A*** and ***C***. An optimal partitioning will minimize the execution time of the algorithm. Let one unit of computation be the multiplication of one row of matrix ***A*** by matrix ***B***, producing one row of the resulting matrix ***C***. The size of this unit of computation does not depend on which rows of matrices ***A*** and ***C*** are involved in the computation. The computational unit will always include n^2 multiplications

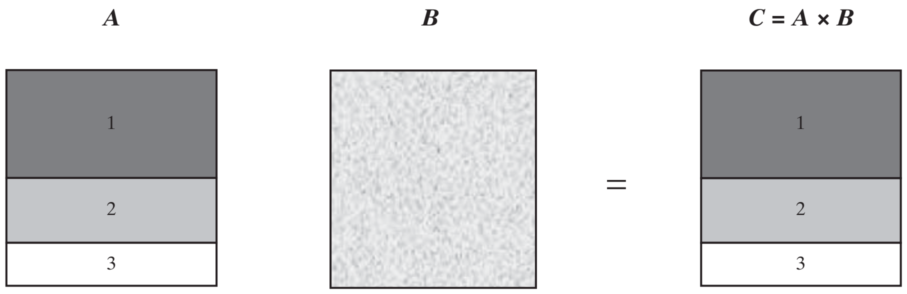

Figure 3.1. Matrix operation $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$ with $n \times n$ matrices ***A***, ***B***, and ***C***. Matrices ***A*** and ***C*** are horizontally sliced such that the number of elements in the slice is proportional to the speed of the processor.

and $n \times (n-1)$ additions. Processor P_i will perform n_i such computation units, where n_i is the number of rows in the slice assigned to this processor, $\Sigma_{i=1}^{p} n_i = n$. Thus, the problem of optimal partitioning of matrices ***A*** and ***C*** is reduced to the problem of optimal distribution of n independent computational units of equal size over p heterogeneous processors of the respective speeds $s_1, \ldots, s_p$, where s_i is the number of rows of matrix ***C*** computed by processor P_i per one time unit. Therefore, we can apply Algorithm 3.1 to solve the partitioning problem.

Note. That straightforward application of Algorithm 3.1 has one disadvantage. In the above example, the size of the computational unit is an increasing function of n. Therefore, the absolute speed of the same processor, measured in computational units per one time unit, will be decreasing with the increase of n, and the application programmer will have to obtain this speed for each particular n used in different runs of the application. At the same time, very often, the relative speed of the processors does not depend on n for quite a wide range of values. Hence, the application programmer could obtain the relative speeds once for some particular n and use the same speeds for other values of n. Actually, nothing prevents us from using relative speeds in this case, in particular, and in Algorithm 3.1, in general. Indeed, minimization of $\frac{n_i+1}{s_i}$ at the refining step of this algorithm will also minimize $\frac{n_i+1}{s_i / \Sigma_{i=1}^{p} s_i}$, as $\Sigma_{i=1}^{p} s_i$ does not depend on i. Therefore, Algorithm 3.1 will return an optimal distribution of computational units, independent on whether we use absolute or relative speeds.

If we reflect on the above application of Algorithm 3.1, we can also make the following observation. If n is big enough and if $p \ll n$, then many straightforward algorithms of refining the distribution obtained after the initialization step will return an approximate solution, which is very close to optimal and satisfactory in practice. For example, the refining could be done by incrementing n_i in a round-robin fashion. Such algorithms return *asymptotically optimal* solutions: The larger the matrix size, the closer the solutions to the optimal ones. One obvious advantage of using modifications of Algorithm 3.1 returning not the exact but the approximate, asymptotically optimal distributions is that the complexity of the distribution algorithms can be reduced to O(p).

To be specific, we have to formalize somehow the notion of an approximate optimal distribution. For example, we can define it as any distribution $n_i\left(\Sigma_{i=1}^{p} n_i = n\right)$ that satisfies the inequality $\left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor \le n_i \le \left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor + 1$. This definition is not perfect because for some combinations of p, n, and s_i, the exact optimal distribution may not satisfy the inequality. Nevertheless, this definition allows us to mathematically formulate the problem of the approximate optimal distribution of independent equal units of computation over a set of heterogeneous processors as follows. Given n independent units of computations, each of equal size, distribute these units of work over $p(p \ll n)$ physical processors $P_1, P_2, \ldots, P_p$ of respective speeds $s_1, s_2, \ldots, s_p$ so that

- The number of computational units n_i assigned to processor P_i shall be approximately proportional to its speed, namely, $\left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor \le n_i \le \left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor + 1$
- $\Sigma_{i=1}^{p} n_i = n$

3.2 DATA DISTRIBUTION PROBLEMS WITH CONSTANT PERFORMANCE MODELS OF HETEROGENEOUS PROCESSORS

In the previous section, the problem of distribution of units of computations in proportion to the speed of heterogeneous processors during multiplication of two dense square matrices was first reduced to the problem of partitioning a matrix and, in the end, to the problem of partitioning a set. This is typical in the design of heterogeneous parallel algorithms when the problem of distribution of computations in proportion to the speed of processors is reduced to the problem of partitioning some mathematical objects such as sets, matrices, graphs, and so on.

In a generic form, a typical partitioning problem with a constant performance model of heterogeneous processors can be formulated as follows:

- Given a set of p processors $P_1, P_2, \ldots, P_p$, the speed of each of which is characterized by a positive constant, s_i
- Partition a mathematical object of the size n (the number of elements in a set or matrix, or the number of nodes in a graph) into p subobjects of the same type (a set into subsets, a matrix into submatrices, a graph into subgraphs, etc.) so that
  - There is one-to-one mapping between the partitions and the processors
  - The size n_i of each partition is approximately proportional to the speed of the processor owing the partition, $\frac{n_i}{s_i} \approx const$
    - That is, it is assumed that the volume of computation is proportional to the size of the mathematical object
    - The notion of approximate proportionality is supposed to be defined for each particular problem; if it is not defined, it means that any partitioning consists of partitions, the sizes of which are approximately proportional to the speeds of the processors owing the partitions

  - The partitioning satisfies some additional restrictions on the relationship between the partitions
    - For example, the submatrices of the matrix may be required to form a two-dimensional $r \times q$ arrangement, where r and q may be either given constants or the parameters of the problem, the optimal value of which should be also found
  - The partitioning minimizes some functional(s), which is(are) used to estimate each partitioning
    - For example, it minimizes the sum of the perimeters of the rectangles representing the submatrices (intuitively, this functional estimates the volume of communications for some parallel algorithms)

The problem of optimal distribution of independent equal computational units presented in Section 3.1 can be formulated as the following instantiation of the generic partitioning problem:

- Given a set of p processors $P_1, P_2, \ldots, P_p$, the speed of each of which is characterized by a positive constant, s_i
- Partition a set of n elements into p subsets so that
  - There is one-to-one mapping between the partitions and the processors
  - The number of elements n_i in each partition is approximately proportional to s_i, the speed of the processor owing the partition, so that $\left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor \le n_i \le \left\lfloor \frac{s_i}{\Sigma_{i=1}^{p} s_i} \times n \right\rfloor + p$
  - The partitioning minimizes $\max_i \frac{n_i}{s_i}$

Another important set partitioning problem is formulated as follows:

- Given a set of p processors $P_1, P_2, \ldots, P_p$, the speed of each of which is characterized by a positive constant, s_i
- Given a set of n unequal elements, the weight of each of which is characterized by a positive constant
- Partition the set into p subsets so that
  - There is one-to-one mapping between the partitions and the processors
  - The total weight of each partition, w_i, is approximately proportional to s_i, the speed of the processor owing the partition
  - The partitioning minimizes $\max_i \frac{w_i}{s_i}$

REFERENCES

Achalakul T and Taylor S. (2003). A distributed spectral-screening PCT algorithm. *Journal of Parallel and Distributed Computing* **63**(3):373–384.

Alexandrov A, Ionescu M, Schauser K, and Scheiman C. (1995). LogGP: Incorporating long messages into the LogP model. *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 24–26, 1995, Santa Barbara, CA; ACM, New York, pp. 95–105.

Barbosa J, Tavares J, and Padilha A. (2000). Linear algebra algorithms in a heterogeneous cluster of personal computers. *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, May 1, 2000, Cancun, Mexico. IEEE Computer Society Press, pp. 147–159.

Bazterra V, Cuma M, Ferraro M, and Facelli J. (2005). A general framework to understand parallel performance in heterogeneous clusters: Analysis of a new adaptive parallel genetic algorithm. *Journal of Parallel and Distributed Computing* **65**(1):48–57.

Beaumont O, Boudet V, Petitet A, Rastello F, and Robert Y. (2001a). A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). *IEEE Transactions on Computers* **50**(10):1052–1070.

Beaumont O, Boudet V, Rastello F, and Robert Y. (2001b). Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems* **12**(10):1033–1051.

Beaumont O, Boudet V, Rastello F, and Robert Y. (2001c). Heterogeneous matrix-matrix multiplication or partitioning a square into rectangles: NP-completeness and approximation algorithms. *Proceedings of the 9th Euromicro Workshop on Parallel and Distributed Processing (PDP 2001)*, February 7–9, 2001, Mantova, Italy, IEEE Computer Society, pp. 298–302.

Becker B and Lastovetsky A. (2006). Matrix multiplication on two interconnected processors. *Proceedings of the 8th IEEE International Conference on Cluster Computing (Cluster 2006)*, September 25–28, 2006, Barcelona, Spain; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Becker B and Lastovetsky A. (2007). Towards data partitioning for parallel computing on three interconnected clusters. *Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, July 5–8, 2007, Hagenberg, Austria, IEEE Computer Society, pp. 285–292.

Bernaschi M and Iannello G. (1998). Collective communication operations: Experimental results vs. theory. *Concurrency: Practice and Experience* **10**(5):359–386.

Bertschinger E. (1995). COSMICS: Cosmological initial conditions and microwave anisotropy codes. *ArXiv Astrophysics e-prints*, http://arxiv.org/abs/astro-ph/9506070.

Boulet P, Dongarra J, Rastello F, Robert Y, and Vivien F. (1999). Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters* **9**(2):197–213.

Brady T, Konstantinov E, and Lastovetsky A. (2006). SmartNetSolve: High level programming system for high performance grid computing. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 25–29, 2006, Rhodes, Greece; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Brady T, Guidolin M, and Lastovetsky A. (2008). Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model. *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid 2008)*, September 29–October 1, 2008, Tsukuba, Japan, IEEE Computer Society, pp. 49–56.

Brightwell R, Fisk L, Greenberg D, Hudson T, Levenhagen M, Maccabe A, and Riesen R. (2000). Massively parallel computing using commodity components. *Parallel Computing* **26**(2–3):243–266.

Buttari A, Dongarra J, Langou J, Langou J, Luszczek P, and Kurzak J. (2007). Mixed precision iterative refinement techniques for the solution of dense linear systems *International Journal of High Performance Computing Applications* **21**(4):457–466.

Canon L-C and Jeannot E. (2006). Wrekavoc: A tool for emulating heterogeneity. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 25–29, 2006, Rhodes, Greece; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Caron E and Desprez F. (2006). DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications* **20**(3):335–352.

Carriero N, Gelernter D, Mattson T, and Sherman A. (1994). The Linda alternative to message-passing systems. *Parallel Computing* **20**(4):633–655.

Casanova H. (2005). Network modeling issues for grid application scheduling. *International Journal of Foundations of Computer Science* **16**(2):145–162.

Casanova H and Dongarra J. (1996). NetSolve: A network server for solving computational science problems. *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, November 17–22, 1996, Pittsburgh, PA; Washington, DC, CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Chamberlain R, Chace D, and Patil A. (1998). How are we doing? An efficiency measure for shared, heterogeneous systems. *Proceedings of the ISCA 11th International Conference on Parallel and Distributed Computing Systems*, September 2–4, 1998, Chicago, IL, pp. 15–21.

Chang C-I. (2003). *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. New York: Kluwer.

Chen Y and Sun X-H. (2006). STAS: A scalability testing and analysis system. *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, September 25–28, 2006, Barcelona, Spain; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Chetverushkin B, Churbanova N, Lastovetsky A, and Trapeznikova M. (1998). Parallel simulation of oil extraction on heterogeneous networks of computers. *Proceedings of the 1998 Conference on Simulation Methods and Applications (CSMA'98)*, November 1–3, 1998, Orlando, FL, Society for Computer Simulation, pp. 53–59.

Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, and Whaley R. (1996a). The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* **5**(3):173–184.

Choi J, Dongarra J, Ostrouchov S, Petitet A, Walker D, and Whaley R. (1996b). A proposal for a set of parallel basic linear algebra subprograms. *Proceedings of the Second International Workshop on Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science (PARA'95)*, August 21–24, 1995, Lyngby, Denmark; Berlin, Germany, Lecture Notes in Computer Science, vol. 1041, Springer, pp. 107–114.

ClearSpeed. (2008). http://www.clearspeed.com/.

Colella P and Woodward P. (1984). The piecewise parabolic method (PPM) for gas-dynamical simulations. *Journal of Computational Physics* **54**(1):174–201.

Crandall P and Quinn M. (1995). Problem decomposition for non-uniformity and processor heterogeneity. *Journal of the Brazilian Computer Society* **2**(1):13–23.

Cuenca J, Giménez D, and Martinez J-P. (2005). Heuristics for work distribution of a homogeneous parallel dynamic programming scheme on heterogeneous systems. *Parallel Computing* **31**(7):711–730.

Culler D, Karp R, Patterson D, Sahay A, Schauser KE, Santos E, Subramonian R, von Eicken T. (1993). LogP: Towards a realistic model of parallel computation. *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 19–22, 1993, San Diego, CA; ACM, New York, pp. 1–12.

Demmel J, Dongarra J, Parlett B, Kahan W, Gu M, Bindel D, Hida Y, Li X, Marques O, Riedy E, Vömel C, Langou J, Luszczek P, Kurzak J, Buttari A, Langou J, and Tomov S. (2007). For Prospectus for the Next LAPACK and ScaLAPACK Libraries. Department of Computer Science, University of Tennessee. *Tech. Rep. UT-CS-07-592*.

Deshpande A and Schultz M. (1992). Efficient parallel programming with Linda. *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, November 16–20, 1992, Minneapolis, MN; Washington, DC, CD-ROM/Abstracts Proceedings, IEEE Computer Society, pp. 238–244.

Dhodhi M, Saghri J, Ahmad I, and Ul-Mustafa R. (1999), D-ISODATA: A distributed algorithm for unsupervised classification of remotely sensed data on network of workstations. *Journal of Parallel and Distributed Computing* **59**(2):280–301.

Dongarra J and Whaley R. (1995). A User's Guide to the BLACS v1.0. Department of Computer Science, University of Tennessee. *Tech. Rep. UT-CS-95-281*.

Dongarra J, Croz J, Duff I, and Hammarling S. (1990). A set of level-3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* **16**(1):1–17.

Dongarra J, van de Geijn R, and Walker D. (1994). Scalability issues affecting the design of a dense linear algebra library. *Journal of Parallel and Distributed Computing* **22**(3):523–537.

Dongarra J, Prylli L, Randriamaro C, and Tourancheau B. (1995). Array Redistribution in ScaLAPACK Using PVM. Department of Computer Science, University of Tennessee. *Tech. Rep. UT-CS-95-310*.

Dorband J, Palencia J, and Ranawake U. (2003). Commodity computing clusters at goddard space flight center. *Journal of Space Communication* **1**(3):23–35.

Dovolnov E, Kalinov A, and Klimov S. (2003). Natural block data decomposition for heterogeneous clusters. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing (IPDPS 2003)*, April 22–26, 2003, Nice, France; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Drozdowski M and Wolniewicz P. (2003). Out-of-core divisible load processing. *IEEE Transactions on Parallel and Distributed Systems* **14**(10):1048–1056.

Foster I. (2002). What is the grid: A three point checklist. *GridToday*, July 20, 2002, http://www.mcs.anl.gov/~itf/Articles/WhatIsTheGrid.pdf.

Foster I, Kesselman C, Nick J, and Tuecke S. (2002). *The physiology of the grid: An open grid services architecture for distributed systems integration*. http://www.globus.org/ogsa.

Fukushige T, Taiji M, Makino J, Ebisuzaki T, and Sugimoto D. (1996). A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: MD-GRAPE. *Astrophysical Journal* **468**: 51–61.

Gabriel E, Fagg G, Bosilca G, Angskun T, Dongarra J, Squyres J, Sahay V, Kambadur P, Barrett B, Lumsdaine A, Castain R, Daniel D, Graham R, and Woodall T. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Proceedings of EuroPVM/MPI 2004)*, Lecture Notes in Computer Science, vol. 3241, (eds. D Kranzlmüller, P Kacsuk, and J Dongarra) Berlin, Germany: Springer, pp. 97–104.

Garey M and Johnson D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Miller Freeman.

van de Geijn R and Watts J. (1997). SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* **9**(4):255–274.

Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, and Sunderam V. (1994). *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press.

Gheller C, Pantano O, and Moscardini L. (1998). A cosmological hydrodynamic code based on the piecewise parabolic method. *Royal Astronomical Society, Monthly Notices* **295**(3):519–533.

Globus. (2008). http://www.globus.org/.

Graham R, Shipman G, Barrett B, Castain R, Bosilca G, and Lumsdaine A. (2006). Open MPI: A high-performance, heterogeneous MPI. *Proceedings of the 8th IEEE International Conference on Cluster Computing (Cluster 2006)*, September 25–28, 2006, Barcelona, Spain; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Grama A, Gupta A, and Kumar V. (1993). Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel & Distributed Technology* **1**(3):12–21.

GridToday. (2004). http://www.on-demandenterprise.com/features/grid_computing_--_hype_or_tripe__07-29-2008_08_06_35.html.

Gropp W, Lusk E, Ashton D, Balaji P, Buntinas D, Butler R, Chan A, Krishna J, Mercier G, Ross R, Thakur R, and Toonen B. (2007). *MPICH2 User's Guide.*

Version 1.0.6. Argonne, IL. Mathematics and Computer Science Division, Argonne National Laboratory.

Grove D and Coddington P. (2001). Precise MPI performance measurement using MPIBench. *Proceedings of HPC Asia*, September 24–28, 2001, Gold Coast, Queensland, Australia, pp. 24–28

Guidolin M and Lastovetsky A. (2008). Grid-Enabled Hydropad: A Scientific Application for Benchmarking GridRPC-based Programming Systems. School of Computer Science and Informatics, University College Dublin. *Tech. Rep. UCD-CSI-2008-10*.

Gupta R and Vadhiyar A. (2007). An efficient MPI_Allgather algorithm for grids. *Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC-16)*, June 25–29, 2007, Monterey, CA, IEEE Computer Society, pp. 169–178.

Gustafson J, Montry G, and Benner R. (1988). Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing* **9**(4):609–638.

HeteroScaLAPACK. (2008). Heterogeneous ScaLAPACK software (HeteroScaLAPACK). School of Computer Science and Informatics, University College Dublin. http://hcl.ucd.ie/project/HeteroScaLAPACK.

Higgins R and Lastovetsky A. (2005). Scheduling for heterogeneous networks of computers with persistent fluctuation of load. *Proceedings of the 13th International Conference on Parallel Computing (ParCo 2005)*, John von Neumann Institute for Computing Series, vol. 33, September 13–16, 2005, Malaga, Spain: Central Institute for Applied Mathematics, pp. 171–178.

High Performance Fortran Forum. (1997). *High Performance Fortran Language Specification (Version 2.0)*. Houston, TX: High Performance Fortran Forum, Rice University.

Hockney R. (1994). The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing* **20**(3):389–398.

Hockney R and Eastwood J. (1981). *Computer Simulation Using Particles*. New York: McGraw Hill.

Ibarra O and Kim C. (1977). Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM* **24**(2):280–289.

Intel. (2004). *Intel MPI Benchmarks. User Guide and Methodology Description*. Bruhl, Germany: Intel GmbH.

IEEE. (1985). *ANSI/IEEE Standard for Binary Floating Point Arithmetic: Std 754-1985*. New York: IEEE Press.

IEEE. (1987). *ANSI/IEEE Standard for Radix Independent Floating Point Arithmetic: Std 854-1987*. New York: IEEE Press.

IEEE. (1994). *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors: Std 1596.5-1993*. New York: IEEE Press.

Gschwind M, Hofstee P, Flachs B, Hopkins M, Watanabe Y, Yamazaki T. (2006). Synergistic processing in Cell's multicore architecture. *IEEE Micro* **26**(2):10–24.

Kaddoura M, Ranka S, and Wang A. (1996). Array decompositions for nonuniform computational environments. *Journal of Parallel and Distributed Computing* **36**(2):91–105.

Kalinov A. (2006). Scalability of heterogeneous parallel systems. *Programming and Computer Software* **32**(1):1–7.

Kalinov A and Klimov S. (2005). Optimal mapping of a parallel application processes onto heterogeneous platform. *Proceedings of 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, April 4–8, 2005, Denver, CO; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Kalinov A and Lastovetsky A. (1999a). Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. *Proceedings of the 7th International Conference on High Performance Computing and Networking Europe (HPCN'99)*, Lecture Notes in Computer Science, vol. 1593, April 12–14, 1999, Amsterdam, The Netherlands; Berlin, Germany, Springer, pp. 191–200.

Kalinov A and Lastovetsky A. (1999b). mpC + ScaLAPACK = Efficient solving linear algebra problems on heterogeneous networks. *Proceedings of the 5th International Euro-Par Conference (Euro-Par'99)*, Lecture Notes in Computer Science, vol. 1685, August 31–September 3, 1999, Toulouse, France; Berlin, Germany, Springer, pp. 1024–1031.

Kalinov A and Lastovetsky A. (2001). Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing* **61**(4):520–535.

Kalluri S, Zhang Z, JaJa J, Liang S, and Townshend J. (2001). Characterizing land surface anisotropy from AVHRR data at a global scale using high performance computing. *International Journal of Remote Sensing* **22**(11):2171–2191.

Karp A and Platt H. (1990). Measuring parallel processor performance. *Communications of the ACM* **22**(5):539–543.

Kielmann T, Bal H, and Verstoep K. (2000). Fast measurement of LogP parameters for message passing platforms. *Proceedings of IPDPS 2000 Workshops*, Lecture Notes in Computer Science, vol. 1800, May 1–5, 2000, Cancun, Mexico; Berlin, Germany, Springer, pp. 1176–1183.

Kishimoto Y and Ichikawa S. (2004). An execution-time estimation model for heterogeneous clusters. *Proceedings of 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 26–30, 2004, Santa Fe, NM; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Kumar S, Chao H, Alamasi G, and Kale L. (2006). Achieving strong scaling with NAMD on Blue Gene/L. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 25–29, 2006, Rhodes, Greece; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Kumar V, Grama A, Gupta A, and Karypis G. (1994). *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA: Benjamin-Cummings and Addison-Wesley.

Kwok Y-K and Ahmad I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys* **31**(4):406–471.

Lastovetsky A. (2002). Adaptive parallel computing on heterogeneous networks with mpC. *Parallel Computing* **28**(10):1369–1407.

Lastovetsky A. (2003). *Parallel Computing on Heterogeneous Networks*. Hoboken, NJ: Wiley-Interscience.

Lastovetsky A. (2006). Scientific programming for heterogeneous systems—Bridging the gap between algorithms and applications. *Proceedings of the 5th International Symposium on Parallel Computing in Electrical Engineering (PARELEC 2006)*, September 13–17, 2006, Bialystok, Poland, IEEE Computer Society, pp. 3–8.

Lastovetsky A. (2007). On grid-based matrix partitioning for heterogeneous processors. *Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, July 5–8, 2007, Hagenberg, Austria, IEEE Computer Society, pp. 383–390.

Lastovetsky A and O'Flynn M. (2007). A performance model of many-to-one collective communications for parallel computing. *Proceedings of the 21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*, March 26–30, 2007, Long Beach, CA; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Lastovetsky A and Reddy R (2004a). Data partitioning with a realistic performance model of networks of heterogeneous computers. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 26–30, 2004, Santa Fe, NM; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Lastovetsky A and Reddy R. (2004b). On performance analysis of heterogeneous parallel algorithms. *Parallel Computing* **30**(11):1195–1216.

Lastovetsky A and Reddy R. (2005). Data partitioning for multiprocessors with memory heterogeneity and memory constraints. *Scientific Programming* **13**(2):93–112.

Lastovetsky A and Reddy R. (2006). HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing* **66**(2):197–220.

Lastovetsky A and Reddy R. (2007a). A novel algorithm of optimal matrix partitioning for parallel dense factorization on heterogeneous processors. *Proceedings of the 9th International Conference on Parallel Computing Technologies (PaCT-2007)*, Lecture Notes in Computer Science, vol. 4671, September 3–7, 2007, Pereslavl-Zalessky, Russia; Berlin, Germany, Springer, pp. 261–275.

Lastovetsky A and Reddy R. (2007b). Data partitioning with a functional performance model of heterogeneous processors. *International Journal of High Performance Computing Applications* **21**(1):76–90.

Lastovetsky A and Reddy R. (2007c). Data partitioning for dense factorization on computers with memory heterogeneity. *Parallel Computing* **33**(12):757–779.

Lastovetsky A and Rychkov V. (2007). Building the communication performance model of heterogeneous clusters based on a switched network. *Proceedings of the 2007 IEEE International Conference on Cluster Computing (Cluster 2007)*, September 17–20, 2007, Austin, TX, IEEE Computer Society, pp. 568–575.

Lastovetsky A and Twamley J. (2005). Towards a realistic performance model for networks of heterogeneous computers. In: *High Performance Computational Science and Engineering (Proceedings of IFIP TC5 Workshop, 2004 World Computer Congress)* (eds. MK Ng, A Doncescu, LT Yang, and T Leng). Berlin, Germany: Springer, pp. 39–58.

Lastovetsky A, Mkwawa I, and O'Flynn M. (2006). An accurate communication model of a heterogeneous cluster based on a switch-enabled Ethernet network. *Proceedings of the 12th International Conference on Parallel and Distributed Systems*

(ICPADS 2006), July 12–15, 2006, Minneapolis, MN, IEEE Computer Society, pp. 15–20.

Lastovetsky A, Reddy R, and Higgins R. (2006). Building the functional performance model of a processor. *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC'06)*, April 23–27, 2006, Dijon, France, ACM Press, pp. 746–753.

Lastovetsky A, O'Flynn M, and Rychkov V. (2007). Optimization of collective communications in heterompi. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Proceedings of EuroPVM/MPI 2007)*, Lecture Notes in Computer Science, vol. 4757, (eds. F Cappello, T Herault, and J Dongarra). Berlin, Germany: Springer, pp. 135–143.

Lastovetsky A, O'Flynn M, and Rychkov V. (2008). MPIBlib: Benchmarking MPI communications for parallel computing on homogeneous and heterogeneous clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Proceedings of EuroPVM/MPI 2008)*, Lecture Notes in Computer Science, vol. 5205, (eds. A Lastovetsky, T Kechadi and J Dongarra) Berlin, Germany: Springer, pp. 227–238.

Le Moigne J, Campbell W, and Cromp R. (2002). An automated parallel image registration technique based on the correlation of wavelet features. *IEEE Transactions on Geoscience and Remote Sensing* **40**(8):1849–1864.

Lee C, Matsuoka S, Talia De, Sussman A, Mueller M, Allen G, and Saltz J. (2001). *A Grid Programming Primer*. Global Grid Forum. http://www.cct.lsu.edu/~gallen/Reports/GridProgrammingPrimer.pdf.

Mazzeo A, Mazzocca N, and Villano U. (1998). Efficiency measurements in heterogeneous distributed computing systems: from theory to practice. *Concurrency: Practice and Experience* **10**(4):285–313.

Message Passing Interface Forum. (1995). *MPI: A Message-passing Interface Standard, ver. 1.1*. University of Tennessee: Knoxville, TN.

MPI. (1994). MPI: A message-passing interface standard. *International Journal of Supercomputer Applications* **8**(3/4):159–416.

Nakada H, Sato M, and Sekiguchi S. (1999). Design and implementations of Ninf: Towards a global computing infrastructure. *Future Generation Computing Systems* **15**(5–6):649–658.

Ohtaki Y, Takahashi D, Boku T, and Sato M. (2004). Parallel implementation of Strassen's matrix multiplication algorithm for heterogeneous clusters. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 26–30, 2004, Santa Fe, NM; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Pastor L and Bosque J. (2001). An efficiency and scalability model for heterogeneous clusters. *Proceedings of the 2001 IEEE International Conference on Cluster Computing*, October 8–11, 2001, Newport Beach, CA: IEEE Computer Society, pp. 427–434.

Pjesivac-Grbovic J, Angskun T, Bosilca G, Fagg G, Gabriel E, and Dongarra J. (2007). Performance analysis of MPI collective operation. *Cluster Computing* **10**(2):127–143.

Plaza A. (2007). Parallel techniques for information extraction from hyperspectral imagery using heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing* **68**(1):93–111.

Plaza A and Chang C-I. (eds.) (2007). *High-Performance Computing in Remote Sensing*. Boca Raton, FL: Chapman & Hall/CRC Press.

Plaza A, Martinez P, Plaza J, and Perez R. (2002). Spatial-spectral endmember extraction by multidimensional morphological operations. *IEEE Transactions on Geoscience and Remote Sensing* **40**(9):2025–2041.

Plaza A, Martinez P, Plaza J, and Perez R. (2005). Dimensionality reduction and classification of hyperspectral image data using sequences of extended morphological transformations. *IEEE Transactions on Geoscience and Remote Sensing* **43**(3): 466–479.

Plaza A, Plaza J, and Valencia D. (2006). AMEEPAR: Parallel morphological algorithm for hyperspectral image classification on heterogeneous networks of workstations. *Proceedings of the 6th International Conference on Computational Science (ICCS 2006)*, Lecture Notes in Computer Science, vol. 3993, May 28–31, 2006, Reading, UK; Berlin, Germany, Springer, pp. 24–31.

Plaza A, Valencia D, Plaza J, and Martinez P. (2006). Commodity cluster-based parallel processing of hyperspectral imagery. *Journal of Parallel and Distributed Computing* **66**(3):345–358.

Plaza A, Plaza J, and Valencia D. (2007). Impact of platform heterogeneity on the design of parallel algorithms for morphological processing of high-dimensional image data. *The Journal of Supercomputing* **40**(1):81–107.

Prylli L and Tourancheau B. (1996). Efficient block cyclic data redistribution. *Proceedings of the Second International Euro-Par Conference on Parallel Processing (EUROPAR'96)*, Lecture Notes in Computer Science, vol 1123, August 26–29, 1996, Lyon, France; Berlin, Germany, Springer, pp. 155–164.

Rabenseifner R. (1999). Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. *Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99)*, September 26–29, 1999, Barcelona, Spain; Berlin, Germany, Springer, pp 77–85.

Reddy R and Lastovetsky A. (2006). HeteroMPI + ScaLAPACK: Towards a dense ScaLAPACK on heterogeneous networks of computers. *Proceedings of the 13th IEEE International Conference on High Performance Computing (HiPC 2006)*, Lecture Notes in Computer Science, vol. 4297, December 18–21, 2006, Bangalore, India, Springer, pp. 242–252.

Reddy R, Lastovetsky A, and Alonso P. (2008). Heterogeneous PBLAS: A Set of Parallel Basic Linear Algebra Subprograms for Heterogeneous Computational Clusters. School of Computer Science and Informatics, University College Dublin. *Tech. Rep. UCD-CSI-2008-2*.

Richards J and Jia X. (2005). *Remote Sensing Digital Image Analysis*, 4th ed. Berlin, Germany: Springer.

RIKEN. (2008). http://www.riken.go.jp/engn/index.html.

Rosenberry W, Kenney D, and Fisher G. (1992). *Understanding DCE*. Sebastopol, CA: O'Reilly.

ScaLAPACK. (1997). The ScaLAPACK project. http://www.netlib.org/scalapack/.

Seymour K, Nakada H, Matsuoka S, Dongarra J, Lee C, and Casanova H. (2002). Overview of GridRPC: A remote procedure call API for grid computing. *Proceedings of the Third International Workshop on Grid Computing (Grid 2002)*, Lecture

Notes in Computer Science, vol. 2536, November 18, 2002, Baltimore, MD; Berlin, Germany, Springer, pp. 274–278.

Shirasuna S, Nakada H, Matsuoka S, and Sekiguchi S. (2002). Evaluating Web services based implementations of GridRPC. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, July 24–26, 2002, Edinburgh, Scotland, IEEE Computer Society.

Smarr L and Catlett CE. (1992). Metacomputing. *Communications of the ACM* **35**(6):44–52.

Snir M, Otto S, Huss-Lederman S, Walker D, and Dongarra J. (1996). *MPI: The Complete Reference*. Cambridge, MA: MIT Press.

Soille P. (2003). *Morphological Image Analysis: Principles and Applications*, 2nd ed. Berlin, Germany: Springer.

Spring J, Spring N, and Wolski R. (2000). Predicting the CPU availability of time-shared Unix systems on the computational grid. *Cluster Computing* **3**(4):293–301.

Sterling T, Lusk E, and Gropp W. (2003). *Beowulf Cluster Computing with Linux*. Cambridge, MA: MIT Press.

Sulistio A, Yeo C, and Buyya R. (2004). A taxonomy of computer-based simulations and its mapping to parallel and distributed systems simulation tools. *Software: Practice and Experience* **34**(7):653–673.

Sun X-H and Rover D. (1994). Scalability of parallel algorithm-machine combinations. *IEEE Transactions on Parallel and Distributed Systems* **5**(6):599–613.

Sun X-H, Chen Y, and Wu M. (2005). Scalability of heterogeneous computing. *Proceedings of the 34th International Conference on Parallel Processing*, June 14–17, 2005, Oslo, Norway, IEEE Computer Society, pp. 557–564.

SunSoft. (1993). *The XDR Protocol Specification. Appendix A of "Network Interfaces Programmer's Guide."* SunSoft. http://docs.sun.com/app/docs/doc/801-6741/6i13kh8sg?a=view.

de Supinski B and Karonis N. (1999). Accurately measuring MPI broadcasts in a computational grid. *Proceedings of the Eighth International Symposium on High Performance Distributed Computing*, August 3–6, 1999, Redondo Beach, CA, pp. 29–37.

Tanaka Y, Nakada H, Sekiguchi S, Suzumura T, and Matsuoka S. (2003). Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing* **1**(1):41–51.

Thakur R, Rabenseifner R, and Gropp W. (2005). Optimization of collective communication operations in MPICH. *International Journal of High Performance Computing Applications* **19**(1):49–66.

Tilton J. (2001). Method for implementation of recursive hierarchical segmentation on parallel computers. US Patent Office, Washington, DC. Pending published application 09/839147.

Tilton J. (2007). Parallel implementation of the recursive approximation of an unsupervised hierarchical segmentation algorithm. In *High-Performance Computing in Remote Sensing* (eds. AJ Plaza and C-I Chang) Boca Raton, FL: Chapman & Hall/CRC Press, pp. 97–107.

Turcotte L. (1993). A Survey of Software Environments for Exploiting Networked Computing Resources. Engineering Research Center, Mississippi State University. *Tech. Rep. MSSU-EIRS-ERC-93-2*.

Unicore. (2008). http://unicore.sourceforge.net/.

Vadhiyar S, Fagg G, and Dongarra J. (2000). Automatically tuned collective communications. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, November 4–10, 2000, Dallas, TX, IEEE Computer Society.

Valencia D, Lastovetsky A, O'Flynn M, Plaza A, and Plaza J. (2008). Parallel processing of remotely sensed hyperspectral images on heterogeneous networks of workstations using HeteroMPI. *International Journal of High Performance Computing Applications* **22**(4):386–407.

Vetter J, Alam S, Dunigan T, Fahey M, Roth P, and Worley P. (2006). Early evaluation of the Cray XT3. *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 25–29, 2006, Rhodes, Greece; CD-ROM/Abstracts Proceedings, IEEE Computer Society.

Wang P, Liu K, Cwik T, and Green R. (2002). MODTRAN on supercomputers and parallel computers. *Parallel Computing* **28**(1):53–64.

Whaley R, Petitet A, and Dongarra J. (2001). Automated empirical optimization of software and the ATLAS Project. *Parallel Computing* **27**(1–2):3–25.

Worsch T, Reussner R, and Augustin W. (2002). On Benchmarking Collective MPI Operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Proceedings of EuroPVM/MPI 2002)*, Lecture Notes in Computer Science, vol. 2474, (eds. D Kranzlmüller, P Kacsuk, J Dongarra, and J Volkert) Berlin: Germany, pp. 271–279.

YarKhan A, Seymour K, Sagi K, Shi Z, and Dongarra J. (2006). Recent developments in GridSolve. *International Journal of High Performance Computing Applications* **20**(1):131–142.

Zhang X and Yan Y. (1995). Modeling and characterizing parallel computing performance on heterogeneous networks of workstations. *Proceedings of the Seventh IEEE Symposium in Parallel and Distributed Processing (SPDPS'95)*, October 25–28, 1995, San Antonio, TX, IEEE Computer Society, pp. 25–34.

Zhang X, Yan Y, and He K. (1994). Latency metric: An experimental method for measuring and evaluating program and architecture scalability. *Journal of Parallel and Distributed Computing* **22**(3):392–410.

Zorbas J, Reble D, and VanKooten R. (1989). Measuring the scalability of parallel computer systems. *Proceedings of the Supercomputing '89*, November 14–18, 1988, Orlando, FL, ACM Press, pp. 832–841.

APPENDICES

APPENDIX A

Appendix to Chapter 3

A.1 PROOF OF PROPOSITION 3.1

Consider an optimal allocation denoted by $o_1, \ldots, o_p$. Let j be such that $\forall i \in \{1, \ldots, p\}, \frac{o_j}{s_j} \geq \frac{o_i}{s_i}$. To prove the correctness of the algorithm, we prove the invariant (I): $\forall i \in \{1, \ldots, p\}, \frac{n_i}{s_i} \leq \frac{o_j}{s_j}$. After the initialization, $n_i \leq \frac{s_i}{\Sigma_{k=1}^{p} s_k} \times n$. We have $n = \Sigma_{k=1}^{p} o_k \leq \frac{o_j}{s_j} \times \Sigma_{k=1}^{p} s_k$. Hence, $\frac{n_i}{s_i} \leq \frac{n}{\Sigma_{k=1}^{p} s_k} \leq \frac{o_j}{s_j}$ and invariant (I) holds. We use an induction to prove that invariant (I) holds after each increment. Suppose that, at a given step, some n_k will be incremented. Before that step, $\Sigma_{i=1}^{p} n_i < n$, hence, there exists $m \in \{1, \ldots, p\}$ such that $n_m < o_k$. We have $\frac{n_m + 1}{s_m} \leq \frac{o_m}{s_m} \leq \frac{o_j}{s_j}$, and the choice of k implies that $\frac{n_k + 1}{s_k} \leq \frac{n_m + 1}{s_m}$. Invariant (I) does hold after the increment. Finally, the time needed to compute the n chunks with the allocation $(n_1, n_2, \ldots, n_p)$ is $\max_i \frac{n_i}{s_i}$, and our allocation is optimal. This proves Proposition 3.1.

A.2 PROOF OF PROPOSITION 3.5

If the algorithm assigns element a_k at each iteration, then the resulting allocation will be optimal by design. Indeed, in this case the distribution of elements over the processors will be produced by the heterogeneous set partitioning (HSP), and hence optimal for each subset $A^{(k)}$.

Consider the situation when the algorithm assigns a group of w ($w > 1$) elements beginning from the element a_k. In that case, the algorithm first produces a sequence of $(w + 1)$ distributions $(n_1^{(k)}, \ldots, n_p^{(k)})$, $(n_1^{(k+1)}, \ldots, n_p^{(k+1)})$, $\ldots, (n_1^{(k+w)}, \ldots, n_p^{(k+w)})$ such that

- the distributions are optimal for subsets $A^{(k)}, A^{(k+1)}, \ldots, A^{(k+w)}$, respectively, and
- $(n_1^{(k)}, \ldots, n_p^{(k)}) > (n_1^{(k+i)}, \ldots, n_p^{(k+i)})$ is only true for $i = w$ (by definition, $(a_1, \ldots, a_p) > (b_1, \ldots, b_p)$ if and only if $(\forall i)(a_i \geq b_i) \wedge (\exists i)(a_i > b_i)$.

Lemma 3.5.1. Let $(n_1, \ldots, n_p)$ and $(n'_1, \ldots, n'_p)$ be optimal distributions such that $n = \Sigma_{i=1}^p n_i > \Sigma_{i=1}^p n'_i = n'$, $(\exists i)(n_i < n'_i)$ and $(\forall j)\left(\max_{i=1}^p \frac{n_i}{s_i} \leq \frac{n_j + 1}{s_j}\right)$. Then, $\max_{i=1}^p \frac{n_i}{s_i} = \max_{i=1}^p \frac{n'_i}{s_i}$.

Proof of Lemma 3.5.1. As $n \geq n'$ and $(n_1, \ldots, n_p)$ and $(n'_1, \ldots, n'_p)$ are both optimal distributions, then $\max_{i=1}^p \frac{n_i}{s_i} \geq \max_{i=1}^p \frac{n'_i}{s_i}$. On the other hand, there exists $j \in [1, p]$ such that $n_j < n'_j$, which implies $n_j + 1 \leq n'_j$. Therefore, $\max_{i=1}^p \frac{n'_i}{s_i} \geq \frac{n'_j}{s_j} \geq \frac{n_j + 1}{s_j}$. As we assumed that $(\forall j)\left(\max_{i=1}^p \frac{n_i}{s_i} \leq \frac{n_j + 1}{s_j}\right)$, then $\max_{i=1}^p \frac{n_i}{s_i} \leq \frac{n_j + 1}{s_j} \leq \frac{n'_j}{s_j} \leq \max_{i=1}^p \frac{n'_i}{s_i}$. Thus, from $\max_{i=1}^p \frac{n_i}{s_i} \geq \max_{i=1}^p \frac{n'_i}{s_i}$ and $\max_{i=1}^p \frac{n_i}{s_i} \leq \max_{i=1}^p \frac{n'_i}{s_i}$, we conclude that $\max_{i=1}^p \frac{n_i}{s_i} = \max_{i=1}^p \frac{n'_i}{s_i}$. *End of proof of Lemma 3.5.1.*

We can apply Lemma 3.5.1 to the pair $(n_1^{(k)}, \ldots, n_p^{(k)})$ and $(n_1^{(k+l)}, \ldots, n_p^{(k+l)})$ for any $l \in [1, w - 1]$. Indeed, $\Sigma_{i=1}^p n_i^{(k)} > \Sigma_{i=1}^p n_i^{(k+l)}$ and $(\exists i)(n_i^{(k)} < n_i^{(k+l)})$. Finally, the HSP guarantees that $(\forall j)\left(\max_{i=1}^p \frac{n_i^{(k)}}{s_i} \leq \frac{n_j^{(k)} + 1}{s_j}\right)$ (see Boulet *et al.*, 1999; Beaumont *et al.*, 2001a). Therefore, $\max_{i=1}^p \frac{n_i^{(k)}}{s_i} = \max_{i=1}^p \frac{n_i^{(k+1)}}{s_i} = \ldots = \max_{i=1}^p \frac{n_i^{(k+w-1)}}{s_i}$. In particular, this means that for any $(m_1, \ldots, m_p)$ such that $\min_{j=k}^{k+w-1} n_i^{(j)} \leq m_i \leq \max_{j=k}^{k+w-1} n_i^{(j)}$ $(i = 1, \ldots, p)$, we will have $\max_{i=1}^p \frac{m_i}{s_i} = \max_{i=1}^p \frac{n_i^{(k)}}{s_i}$. The allocations made in the end by the Reverse algorithm for the elements $a_k, a_{k+1}, \ldots, a_{k+w-1}$ result in a new sequence of distributions for subsets $A^{(k)}, A^{(k+1)}, \ldots, A^{(k+w-1)}$ such that each next distribution differs from the previous one for exactly one processor. Each distribution $(m_1, \ldots, m_p)$ in this new sequence satisfies the inequality $\min_{j=k}^{k+w-1} n_i^{(j)} \le m_i \le \max_{j=k}^{k+w-1} n_i^{(j)}$ $(i = 1, \ldots, p)$. Therefore, they will all have the same cost $\max_{i=1}^{p} \frac{n_i^{(k)}}{s_i}$, which is the cost of the optimal distribution for these subsets found by the HSP. Hence, each distribution in this sequence will be optimal for the corresponding subset. This proves Proposition 3.5.

APPENDIX B

Appendix to Chapter 4

B.1 PROOF OF PROPOSITION 4.1

First, we formulate a few obvious properties of the functions $s_i(x)$.

Lemma 4.1. The functions $s_i(x)$ are bounded.

Lemma 4.2. Any straight line coming through the origin of the coordinate system intersects the graph of the function $s_i(x)$ in no more than one point.

Lemma 4.3. Let $x_i^{(M_k)}$ be the coordinate of the intersection point of $s_i(x)$ and a straight line M_k coming through the origin of the coordinate system ($k \in \{1,2\}$). Then $x_i^{(M_1)} \geq x_i^{(M_2)}$ if and only if $\angle(M_1,X) \leq \angle(M_2,X)$, where $\angle(M_k,X)$ denotes the angle between the line M_k and the x-axis.

Since $s_i(x)$ are continuous and bounded, the initial lines U and L always exist. Since there is no more than one point of intersection of the line L with each of $s_i(x)$, L will make a positive angle with the x-axis. Thus, both U and L will intersect each $s_i(x)$ exactly in one point. Let $x_i^{(U)}$ and $x_i^{(L)}$ be the coordinates of the intersection points of the U and L with $s_i(x)$ $(1 \leq i \leq p)$, respectively. Then, by design, $\Sigma_{i=1}^p x_i^{(U)} \leq n \leq \Sigma_{i=1}^p x_i^{(L)}$. This invariant will hold after each iteration of the algorithm. Indeed, if line M bisects the angle between lines U and L, then $\angle(L,X) \leq \angle(M,X) \leq \angle(U,X)$. Hence, $\Sigma_{i=1}^p x_i^{(U)} \leq \Sigma_{i=1}^p x_i^{(M)} \leq \Sigma_{i=1}^p x_i^{(L)}$. If $\Sigma_{i=1}^p x_i^{(M)} \leq n$, then $\Sigma_{i=1}^p x_i^{(U)} \leq \Sigma_{i=1}^p x_i^{(M)} \leq n \leq \Sigma_{i=1}^p x_i^{(L)}$ and after Step 4 of the algorithm, $\Sigma_{i=1}^p x_i^{(U)} \leq n \leq \Sigma_{i=1}^p x_i^{(L)}$. If $\Sigma_{i=1}^p x_i^{(M)} \geq n$, then $\Sigma_{i=1}^p x_i^{(U)} \leq n \leq \Sigma_{i=1}^p x_i^{(M)} \leq \Sigma_{i=1}^p x_i^{(L)}$ and after Step 4 of the algorithm, $\Sigma_{i=1}^p x_i^{(U)} \leq n \leq \Sigma_{i=1}^p x_i^{(L)}$. Thus, after each iteration of the algorithm, the "ideal" optimal line O such that $\Sigma_{i=1}^p x_i^{(O)} = n$ will be lying between lines U and L. When the algorithm reaches Step 5, we have $x_i^{(L)} - x_i^{(U)} < 1$ for all $1 \leq i \leq p$, which means that the interval $\left[x_i^{(L)}, x_i^{(U)}\right]$ contains, at most, one integer value. Therefore, either $n_i = \lfloor x_i^{(U)} \rfloor = \lfloor x_i^{(O)} \rfloor$ or $n_i = \lfloor x_i^{(U)} \rfloor = \lfloor x_i^{(O)} \rfloor - 1$. Proposition 4.1 is proved.

B.2 PROOF OF PROPOSITION 4.2

The execution time obtained with allocation (n_1, n_2, …, n_p) is given by $\max_i \frac{n_i}{s_i(n_i)}$. The geometrical interpretation of this formula is as follows. Let M_i be the straight line connecting the points (0,0) and (n_i, $s_\mathrm{i}(n_i)$). Then $\frac{n_i}{s_i(n_i)} = \cot\angle(M_i, X)$. Therefore, minimization of $\max_i \frac{n_i}{s_i(n_i)}$ is equivalent to maximization of $\min_i \angle\,(M_i,X)$. Let $\{S_1, S_2, …\}$ be the set of all straight lines such that

- S_k connects (0,0) and (m, $s_i(m)$) for some $i \in \{1, …, p\}$ and some integer m, and
- S_k lies below M_i for any $i \in \{1, …, p\}$.

Let $\{S_1, S_2, …\}$ be ordered in the decreasing order of $\angle(S_k, X)$. The execution time of the allocation (n_1, n_2, …, n_p) is represented by line M_k such that $\angle(M_k,X) = \min_i \angle\,(M_i,X)$. Any increment of n_i means moving one more line from the set $\{S_1,S_2, …\}$ into the set of lines representing the allocation. At each step of the increment, Algorithm 4.3 moves the line making the largest angle with the x-axis. This means that after each increment, the algorithm gives the optimal allocation (n_1,n_2, …,n_p) under the assumption that the total number of chunks, which should be allocated, is equal to $n_1 + n_2 + … + n_p$ (any other increment gives a smaller angle, and hence, longer execution time). Therefore, after the last increment, the algorithm gives the optimal allocation (n_1,n_2,…,n_p) under the assumption that $n_1 + n_2 + … + n_p = n$. Proposition 4.1 is proved.

B.3 PROOF OF PROPOSITION 4.3

First, we estimate the complexity of one iteration of Algorithm 4.2. At each iteration, we need to find the points of intersection of p graphs $y = s_1(x)$, $y = s_2(x)$, …, $y = s_p(x)$ and a straight line $y = a \times x$. In other words, at each iteration, we need to solve p equations of the form $a \times x = s_i(x)$. As we need the same constant number of operations to solve each equation, the complexity of this part of one iteration will be $O(p)$. The test for stopping (Step 2 of the algorithm) also takes a constant number of operations per function $s_i(x)$, making the complexity of this part of one iteration $O(p)$. Therefore, overall, the complexity of one iteration of Algorithm 4.2 will be $O(p)$.

Next, we estimate the number of iterations of this algorithm. To do it, we use the following lemma that states one important property of the initial lines U and L obtained at the Step 1 of Algorithm 4.2.

Lemma 4.4. Let the functions $s_i(x)$ $(1 \le i \le p)$ satisfy the conditions of Proposition 4.1, and the heterogeneity of processors $P_1, P_2, \ldots, P_p$ be bounded. Let O be the point (0,0), A_i be the point of intersection of the initial line U and $s_i(x)$, and B_i be the point of intersection of the initial line L and $s_i(x)$. Then, there exist constants c_1 and c_2 such that $c_1 \le \frac{OB_i}{OA_i} \le c_2$ for any $i \in \{1,2, \ldots, p\}$.

Proof of Lemma 4.4. The full proof of Lemma 4.4 is technical and very lengthy. Here, we give a relatively compact proof of the lemma under the additional assumption that the functions $s_i(x)$ $(1 \le i \le p)$ are monotonically decreasing. First, we prove that there exist constants c_1 and c_2 such that $c_1 \le \frac{OB}{OA} \le c_2$, where A is the point of intersection of the initial line U and $s_{\max}(x) = \max_i s_i(x)$, and B is the point of intersection of the initial line L and $s_{\max}(x)$ (see Fig. B.1). Since the heterogeneity of the processors $P_1, P_2, \ldots, P_p$ is bounded, there exists a constant c such that $\max_{x \in R_+} \frac{s_{\max}(x)}{s_{\min}(x)} \le c$. In particular, this means that $\frac{BD}{FD} \le c$ and $\frac{AC}{EC} \le c$. Let us prove that $\frac{OB}{OA} \le c$. We have $OB = \sqrt{OD^2 + BD^2}$. Since $\frac{OD}{OC} = \frac{BD}{EC}$, we have $OD = \frac{BD}{EC} \times OC$. Since $s_{\min}(x)$

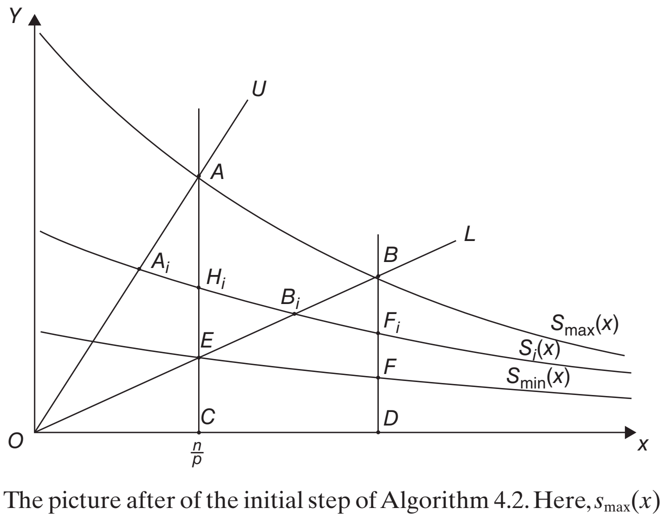

Figure B.1. The picture after of the initial step of Algorithm 4.2. Here, $s_{\max}(x) = \max_i s_i(x)$ and $s_{\min}(x) = \min_i s_i(x)$.

monotonically decreases on the interval $\left[\frac{n}{p}, \infty\right]$, $FD \le EC$, and hence, $\frac{BD}{EC} \le \frac{BD}{FD} \le c$. Thus, $OD \le c \times OC$ and $BD \le c \times EC$. Therefore, $\sqrt{OD^2 + BD^2} \le \sqrt{c^2 + OC^2 + c^2 \times EC^2} = c \times \sqrt{OC^2 + EC^2} = c \times OE$, and hence, $\frac{OB}{OE} \le c$. Since $OA \le OE$, then $\frac{OB}{OA} \le \frac{OB}{OE} \le c$. Next, let us prove that $\frac{OB}{OA} \ge \frac{1}{c}$. We have $OB \ge OE$ and $AC \le c \times EC$. Therefore,

$$\frac{OB}{OA} \ge \frac{OE}{OA} = \frac{\sqrt{OC^2 + EC^2}}{\sqrt{OC^2 + AC^2}} = \frac{OC \times \sqrt{1 + \left(\frac{EC}{OC}\right)^2}}{OC \times \sqrt{1 + c^2 \times \left(\frac{EC}{OC}\right)^2}} = \frac{1}{c} \times \sqrt{\frac{1 + \left(\frac{EC}{OC}\right)^2}{\frac{1}{c^2} + \left(\frac{EC}{OC}\right)^2}}$$

. Since $c \ge 1$, then $\sqrt{\frac{1 + \left(\frac{EC}{OC}\right)^2}{\frac{1}{c^2} + \left(\frac{EC}{OC}\right)^2}} \ge 1$, and hence, $\frac{OB}{OA} \ge \frac{1}{c}$.

Now we are ready to prove Lemma 4.4. We have $\frac{OB_i}{OA_i} \le \frac{OB}{OA_i} = \frac{1}{OA_i} \times OB$. Since $s_i(x)$ is monotonically decreasing, then $\frac{OA}{OA_i} \le \frac{AC}{CH_i}$. Since the heterogeneity of the processors is bounded by the constant c, then $\frac{AC}{CH_i} \le c$. Hence, $\frac{1}{OA_i} \le \frac{c}{OA}$. Therefore, $\frac{OB_i}{OA_i} \le \frac{c}{OA} \times OB = c \times \frac{OB}{OA} \le c^2$. Next, we have $\frac{OB_i}{OA_i} \ge \frac{OB_i}{OA}$. Since $s_i(x)$ is monotonically decreasing, then $\frac{BD}{F_iD} \ge \frac{OB}{OB_i}$. Since the heterogeneity of the processors is bounded by the constant c, then $\frac{BD}{F_iD} \le c$. Therefore, $OB_i \ge \frac{OB}{c}$. Thus, $\frac{OB_i}{OA_i} \ge \frac{OB_i}{OA} \ge \frac{OB}{c \times OA} \ge \frac{1}{c^2}$.

This proves Lemma 4.4.

Bisection of the angle $\angle A_iOB_i$ at the very first iteration will divide the segment A_iB_i of the graph of the function $s_i(x)$ in the proportion $\frac{Q_iB_i}{A_iQ_i} \approx \frac{OB_i}{OA_i}$ (see Fig. B.2). Correspondingly, $\frac{x_i^{(L)} - x_i^{(M)}}{x_i^{(M)} - x_i^{(U)}} \approx \frac{OB_i}{OA_i}$. Since $(b - a)$ approximates the number of integers in the interval $[a, b]$, $\Delta_i = \min\left\{\frac{x_i^{(L)} - x_i^{(M)}}{x_i^{(L)} - x_i^{(U)}}, \frac{x_i^{(M)} - x_i^{(U)}}{x_i^{(L)} - x_i^{(U)}}\right\}$

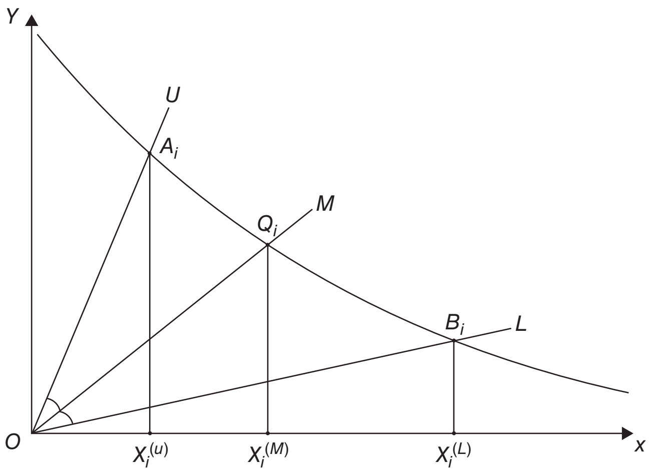

Figure B.2. Bisection of the angle $\angle A_iOB_i$ at the very first iteration into two equal angles. The segment A_iB_i of the graph of the function $s_i(x)$ will be divided in the proportion $\dfrac{Q_iB_i}{A_iQ_i} \approx \dfrac{OB_i}{OA_i}$.

will approximate the lower bound on the fraction of the set $\{\lceil x_i^{(U)} \rceil, \lceil x_i^{(U)} \rceil + 1, \ldots, \lfloor x_i^{(L)} \rfloor\}$ of possible numbers of chunks to be allocated to the processor P_i, which is excluded from consideration after this bisection. Since $c_1 \le \dfrac{OB_i}{OA_i} \le c_2$, then $\Delta_i \ge \dfrac{c_1}{c_2+1} = \Delta$. Indeed, let $q_i = x_i^{(L)} - x_i^{(M)}$ and $r_i = x_i^{(M)} - x_i^{(U)}$. We have $c_1 \le \dfrac{q_i}{r_i} \le c_2$. Therefore, $c_1 \times r_i \le q_i \le c_2 \times r_i$ and $(c_1 + 1) \times r_i \le q_i + r_i \le (c_2 + 1) \times r_i$. Hence, $\dfrac{q_i}{q_i + r_i} \ge \dfrac{c_1 \times r_i}{(c_2+1) \times r_i} = \dfrac{c_1}{c_2+1}$.

$\Delta_i \ge \dfrac{c_1}{c_2+1} = \Delta$ means that after this bisection, at least $\Delta \times 100\%$ of the possible solutions will be excluded from consideration for each processor P_i. The difference in length between OB_i and OA_i will be getting smaller and smaller with each next iteration. Therefore, no less than $\Delta \times 100\%$ of the possible solutions will be excluded from consideration after each iteration of Algorithm 4.2. The number of possible solutions in the initial set for each processor P_i is obviously less than n. The constant Δ does not depend on p or n (actually, this parameter just characterizes the heterogeneity of the set of processors). Therefore, the number of iterations k needed to arrive at the final solution can be found from the equation $(1 - \Delta)^k \times n = 1$, and we have

$k = \frac{1}{\log_2\left(\frac{1}{1-\Delta}\right)} \times \log_2 n$. Thus, overall, the complexity of Algorithm 4.2 will be $O(p \times \log_2 n)$. Proposition 4.3 is proved.

B.4 FUNCTIONAL OPTIMIZATION PROBLEM WITH OPTIMAL SOLUTION, LOCALLY NONOPTIMAL

Consider a simple example with three processors $\{P_1, P_2, P_3\}$ distributing nine columns. Table B.1 shows the functional performance models of the processors $S = \{s_1(x,y), s_2(x,y), s_3(x,y)\}$, where $s_i(x,y)$ is the speed of the update of a $x \times y$ matrix by processor P_i.

Table B.2 shows the distribution of these nine columns, demonstrating that there may be no globally optimal allocation of columns that minimizes the execution time of all steps of the LU factorization.

The first column of Table B.2 represents the step k of the parallel LU factorization. The second column shows the global allocation of columns minimizing the total execution time of LU factorization. The third column shows the execution time of the step k of the LU factorization resulting from this allocation. The execution time $t_i^{(k)}$ for processor P_i needed to update a matrix of size $(9-k) \times n_i^{(k)}$ is calculated as $\frac{V\left(9-k, n_i^{(k)}\right)}{s_i\left(9-k, n_i^{(k)}\right)} = \frac{(9-k) \times n_i^{(k)}}{s_i\left(9-k, n_i^{(k)}\right)}$, where $n_i^{(k)}$

TABLE B.1 Functional Model of Three Processors, P_1, P_2, P_3

Problem sizes (x,y)	$s_1(x,y)$	$s_2(x,y)$	$S_3(x,y)$
(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8)	6, 6, 6, 6, 6, 4, 4, 4	18, 18, 18, 18, 18, 18, 18, 2	18, 18, 18, 18, 18, 18, 18, 2
(2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (2, 8)	6, 6, 6, 6, 5, 4, 3, 3	18, 18, 18, 18, 9, 8, 8, 2	18, 18, 18, 18, 15, 12, 8, 2
(3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8)	6, 6, 6, 5, 4, 3, 3, 3	18, 18, 18, 9, 8, 8, 6, 2	18, 18, 18, 12, 8, 8, 8, 2
(4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4, 6), (4, 7), (4, 8)	6, 6, 5, 4, 3, 3, 3, 3	18, 18, 9, 9, 8, 6, 5, 2	18, 18, 12, 9, 8, 6, 6, 2
(5, 1), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8)	6, 5, 4, 3, 3, 3, 2, 2	18, 9, 8, 8, 6, 5, 3, 1	18, 15, 8, 8, 6, 5, 5, 1
(6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (6, 6), (6, 7), (6, 8)	4, 4, 3, 3, 3, 2, 1, 1	18, 8, 8, 6, 5, 3, 2, 1	18, 12, 8, 6, 5, 3, 3, 1
(7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (7, 8)	4, 3, 3, 3, 2, 1, 1, 1	18, 8, 8, 6, 5, 3, 2, 1	18, 8, 8, 6, 5, 3, 2, 1
(8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (8, 8)	4, 3, 3, 3, 2, 1, 1, 1	2, 2, 2, 2, 1, 1, 1, 1	2, 2, 2, 2, 1, 1, 1, 1

TABLE B.2 Distribution of Nine Column Panels over Three Processors, P_1, P_2, P_3

Step of LU factorization (k)	Global allocation of columns minimizing the overall execution time	Execution time of LU at step k	Local optimal distribution $\{n_1^{(k)}, n_2^{(k)}, n_3^{(k)}\}$ for problem size $(9-k, 9-k)$	Minimum possible execution time for problem size $(9-k, 9-k)$
1	$P_1P_1P_1P_1P_2P_3P_2P_3$	8	{4, 2, 2}	8
2	$\boldsymbol{P_1P_1P_1P_2P_3P_2P_3}$	**7**	**{2, 3, 2}**	$\boldsymbol{\frac{14}{3}}$
3	$\boldsymbol{P_1P_1P_2P_3\ P_2P_3}$	**3**	**{1, 2, 3}**	$\boldsymbol{\frac{3}{2}}$
4	$P_1P_2P_3P_2P_3$	$\frac{10}{9}$	{1, 2, 2}	$\frac{10}{9}$
5	$P_2P_3P_2P_3$	$\frac{4}{9}$	{0, 2, 2}	$\frac{4}{9}$
6	$P_3P_2P_3$	$\frac{1}{3}$	{0, 1, 2}	$\frac{1}{3}$
7	P_2P_3	$\frac{1}{9}$	{0, 1, 1}	$\frac{1}{9}$
8	P_3	$\frac{1}{18}$	{0, 0, 1}	$\frac{1}{18}$
Total execution time of LU factorization		**20**		

denotes the number of columns updated by the processor P_i (formula for the volume of computations explained below). The fourth column shows the distribution of columns, which results in the minimal execution time to solve the problem size $(9-k,9-k)$ at step k of the LU factorization. This distribution is determined by considering all possible mappings and choosing the one that results in minimal execution time. The fifth column shows these minimal execution times for the problem size $(9-k,9-k)$. For example, consider the step $k = 2$, the local optimal distribution resulting in the minimal execution time for the problem size {7, 7} is $\{P_1\ P_1\ P_2\ P_2\ P_2\ P_3\ P_3\}$, the speeds given by the speed functions S shown in Table B.2 are {3, 8, 8}. So the number of columns assigned to processors $\{P_1, P_2, P_3\}$ are {2, 3, 2}, respectively. The execution times are $\left\{\frac{7\times 2}{3}, \frac{7\times 3}{8}, \frac{7\times 2}{8}\right\} = \left\{\frac{14}{3}, \frac{21}{8}, \frac{14}{8}\right\}$. The execution time to solve the problem size {7, 7} is the maximum of these execution times, $\frac{14}{3}$.

Consider again the step $k = 2$ shown in bold in the Table B.2. It can be seen that the global optimal allocation shown in the second column does not result in the minimal execution time for the problem size at this step, which is {7, 7}. The execution time of the LU factorization at this step based on the global optimal allocation is 7, whereas the minimal execution time given by the local optimal distribution for the problem size {7, 7} at this step is $\frac{14}{3}$.

INDEX

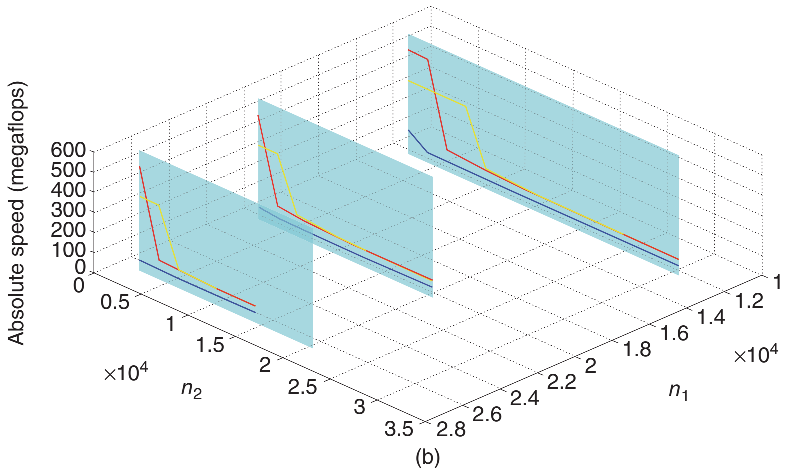

Figure 4.8. (b) Curves on the plane represent the absolute speeds of the processors against variable y, given parameter x is fixed. (See text for full caption.)

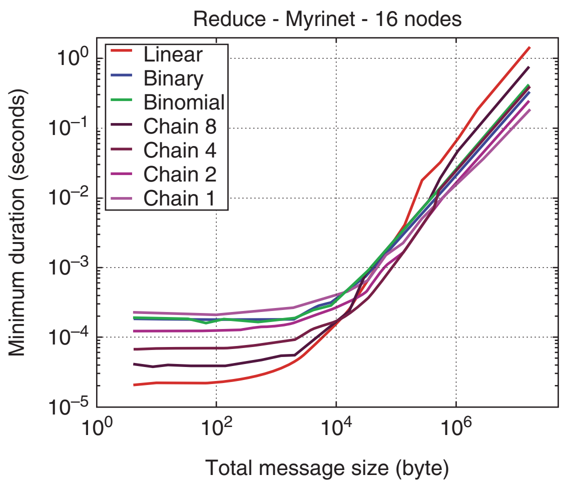

Figure 8.1. Multiple implementations of the MPI reduce operation on 16 nodes.

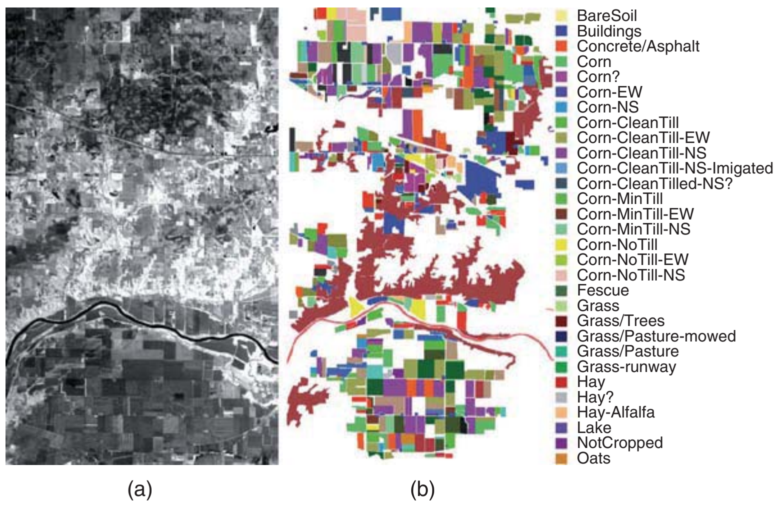

Figure 10.6. (a) Spectral band at 587 nm wavelength of an AVIRIS scene comprising agricultural and forest features at Indian Pines, Indiana. (b) Ground truth map with 30 mutually exclusive classes.

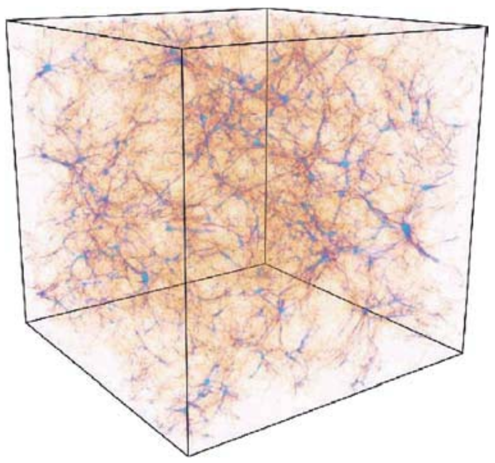

Figure 11.1. Example of a Hydropad output.

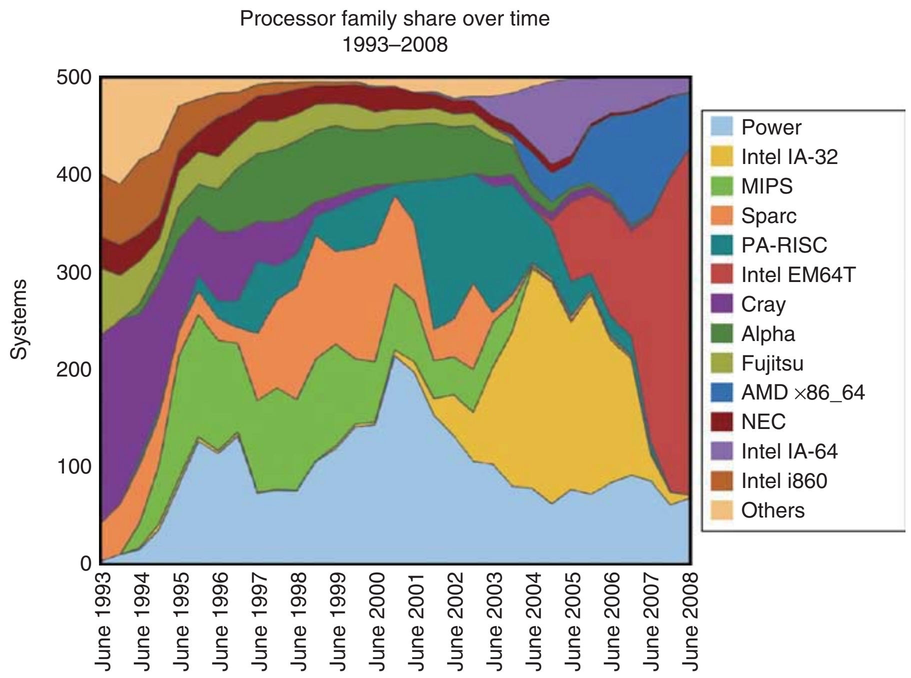

Figure 12.2. Main processor families seen in the TOP500.