Optimization of Data-Parallel Scientific Applications on Highly Heterogeneous Modern HPC Platforms

Ziming Zhong
B.Eng, M.Eng

The thesis is submitted to University College Dublin in fulfilment of the requirements for the degree of Doctor of Philosophy

School of Computer Science and Informatics

Head of School: John Dunnion
Supervisor: Alexey Lastovetsky

August 2014

Abstract

Over the past decade, the design of microprocessors has been shifting to a new model where the microprocessor has multiple homogeneous processing units, aka cores, as a result of heat dissipation and energy consumption issues. Meanwhile, the demand for heterogeneity increases in computing systems due to the need for high performance computing in recent years. The current trend in gaining high computing power is to incorporate specialized processing resources such as manycore Graphic Processing Units in multicore systems, thus making a computing system heterogeneous.

Maximum performance of data-parallel scientific applications on heterogeneous platforms can be achieved by balancing the load between heterogeneous processing elements. Data parallel applications can be load balanced by applying data partitioning with respect to the performance of the platform's computing devices. However, load balancing on such platforms is complicated by several factors, such as contention for shared system resources, non-uniform memory access, limited GPU memory and slow bandwidth of PCIe, which connects the host processor and the GPU.

In this thesis, we present methods of performance modeling and performance measurement on dedicated multicore and multi-GPU systems. We model a multicore and multi-GPU system by a set of heterogeneous abstract processors determined by the configuration of the parallel application. Each abstract processor represents a processing unit made of one or a group of processing elements executing one computational kernel of the application. We group processing units by shared resources, and measure the performance of processing units in each group simultaneously, thereby taking into account the influence of resource contention. We investigate the impact of resource contention, and the impact of process mapping on systems of NUMA architecture on the performance of processing units. Using the proposed method for measuring performance, we built functional performance models of abstract processors, and partition data of data parallel applications using these performance models to balance the workload.

We evaluate the proposed methods with two typical data parallel applications, namely parallel matrix multiplication and numerical simulation of lid-driven cavity flow. Experimental results demonstrate that data partitioning algorithms based on functional performance models built using proposed methods are able to balance the workload of data parallel applications on heterogeneous multicore and multi-GPU platforms.

Acknowledgement

I would like to thank my supervisor Alexey Lastovetsky for giving me the opportunity to do my PhD in the Heterogeneous Computing Laboratory, one of the world research leaders in the field of high performance heterogeneous computing. I am greatly indebted to him for his valuable advice, guidance, and encouragement.

I would like to thank Vladimir Rychkov who helped me a lot through the duration of my study. I would like to thank David Clarke for his help with the Fupermod project. I am also thankful to everyone from the HCL research group, Jun, Jean-Noel, Rob, Kiril, Oleg, Ashley, Ken, Khalid, Tania, Amani. This thesis would have never been possible without your help.

I would like to thank China Scholarship Council and University College Dublin for their financial support with my grant funding. I am also grateful to the financial support of COST Action IC0805 for my visit to University Jaume I and for attending the summer school held in Uppsala University.

I would like to thank the administrative and technical staff of School of Computer Science and Informatics. I am also thankful to all the administrative and technical staff who maintain the Grid5000 infrastructure and the computing resources contained within Innovative Computing Laboratory where I performed most of my experimental work.

I would like to thank the many wonderful friends I met in Ireland who made this journey full of pleasures. I would like to thank all those friends in China who maintain our friendship at a distance for years. Finally I would like to thank my family for supporting me constantly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past six decades, the field of scientific computing has undergone significant changes, including the architecture of underlying platforms and the programming models. The introduction of vector computer systems marked the beginning of modern supercomputing in 1970s. In 1980s, performance was increased primarily by improved chip technologies and by producing shared-memory multiprocessor systems, sometimes referred to as symmetric multiprocessors (SMPs); In 1990s, massively parallel processors (MPPs), which employ a large number of interconnected off-the-shelf processors to perform computations in parallel, became the focus of interest and eventually dominated the top-end of high performance computing. Starting from the beginning of the 20th century, computer clusters became increasingly popular and displaced MPPs a few years later as the dominant type of parallel platforms ever since, as a result of convergence of a number of computing trends including the availability of low cost microprocessors, high speed networks, and software for high performance distributed computing. Programming models for these platforms have also evolved over this time. Whereas most machines relied on custom APIs for messaging and loop-based parallelism, current models standardize these APIs across platforms. Message passing libraries such as MPI, thread libraries such as POSIX threads, and directive based models such as OpenMP are widely accepted as standards, and have been ported to a variety of platforms.

Computational physics applications have been the primary drivers in the development of parallel computing over the last 30 years [42]. Applications are often defined by a set of partial differential equations (PDEs). Attention was focused on the discretization of PDEs and the solution of the linear and nonlinear equations generated by discretization. Significant efforts have been made to facilitate the development of scientific applications for traditional homogeneous parallel platforms such as SMPs and MPPs. Take dense linear algebra as an example, which is heavily used in scientific computing. At the lowest level, BLAS [85], a set of subroutines that perform common linear algebra operations such as vector scaling and matrix product, has been standardized. Highly optimized implementations have been developed by hardware vendors and other authors, like ACML (AMD Core Math Library) [1] and GotoBLAS [52, 53], and are widely used as building blocks in higher-level software packages. At a higher level, LAPACK [4], a set of routines for solving systems of linear equations and linear least squares and for implementing associated matrix factorizations such as LU, QR decomposition and so on, has been developed based on BLAS and become the *de facto* standard for numerical linear algebra on shared-memory multiprocessor systems with a memory hierarchy. At the highest level, ScaLAPACK [28] (or Scalable LAPACK) has extended LAPACK to run on parallel distributed-memory systems. A significant number of software packages have been developed and tuned for solving specific domain problems based on various basic numerical software libraries as above. Similarly, with parallel computing becoming increasingly popular, many well-tuned software packages have also been developed to advance the state-of-the-art in various other scientific disciplines and engineering for widespread parallel platforms.

The mainstream of traditional HPC platforms, including symmetric multiprocessors, massively parallel processors, and computer clusters, is commonly homogeneous. However, the demand for heterogeneity increases in computing systems due to the need for high performance computing in recent years. The design of microprocessors has been shifting to a new model where the microprocessor has multiple homogeneous processing units, aka cores, as a result of heat dissipation and energy consumption issues. In the multicore approach a handful of conventional powerful cores are integrated on the same die, whereas in the manycore approach a significant number of simple and less powerful cores are commonly used focusing more on the execution throughput of parallel applications. The current trend in gaining high computing power is to incorporate specialized processing resources such as Graphic Processing Units (GPUs) in multicore systems, thus making a computing system heterogeneous. This allows a designer to use multiple types of processing elements, each able to perform the tasks that it is best suited for. Furthermore, the heterogeneity in modern computing systems gradually goes down to the chip level where formerly discrete components becomes integrated parts of a system-on-chip. For example, the AMD Fusion Accelerated Processing Unit (APU) integrates processors and graphics on the same die. In summary, modern HPC platforms have been becoming increasingly heterogeneous and hierarchical, which can be exemplified by multicore clusters enhanced with specialized accelerators.

Highly heterogeneous and hierarchical HPC platforms, where multicore processors are coupled with graphics processing units (GPUs), have been widely used in high performance computing as one approach to continuing performance improvement while managing the new challenge of energy efficiency. Over recent years, critical scientific software has been gradually ported to multicore and GPU architectures. Although some legacy software packages and programming languages could be used directly, the introduction of multicores in HPC resulted in redesign of some critical software packages and significant refactoring of some existing parallel applications. For example, the parallel linear algebra for scalable multicore architectures (PLASMA) [3] project has been started aiming to create software frameworks to facilitate application development across a range of multicore architectures. It introduces new algorithms redesigned to maximize data reuse in the cache levels in multicore systems; and relies on runtime scheduling of parallel tasks to fulfill the cores instead of the fork-join paradigm used in legacy LAPACK package. To facilitate development of general-purpose applications on GPUs, new programming models, such as Brook+ [19], CUDA [97, 96], and OpenCL [72], were created. A large number of algorithms and specific applications have been successfully ported to GPUs claiming substantial speedup over their optimized CPU counterparts. The transition from traditional homogeneous platforms to modern hybrid ones is challenging in the aspect of efficient utilization of the heterogeneous hardware and existing optimized software. There has emerged some programming and execution environments for hybrid platforms over recent years, such as Merge [87], StarPU [7], MAGMA [3]. However, more effort still needs to be invested into the research on efficient utilization of hybrid platforms, including efficient workload distribution and load balancing across heterogeneous processing elements.

The standard parallel architectures support a variety of decomposition strategies, such as decomposition by tasks and decomposition by data. Data decomposition is a powerful and commonly used method for deriving concurrency in parallel scientific applications that operate on large data structures such as matrices and graphs. Firstly the data on which the computations are performed is partitioned, then this data partitioning is used to induce a partitioning of the computations into tasks. Once a computation has been decomposed into tasks, these tasks are mapped onto processes to be executed in parallel with the objective that all tasks complete in the shortest amount of elapsed time. For example, on traditional homogeneous parallel platforms, in algorithms that use dense matrices and have structured and regular interaction patterns, the matrices are usually partitioned into a large number of small blocks of the same size; then, these blocks are distributed among processes in a round-robin manner in order to alleviate the load imbalance problem and incur a lower volume of inter-process interaction, also known as block-cyclic distribution. In numerical simulations of physical phenomenon, the problem domain is commonly discretized and represented by a mesh of elements, each generally associated with the same amount of computation. Then, the mesh is partitioned into a number of parts (each assigned to a process) such that each part contains roughly the same number of vertices in order to balance the load, and the number of edges that cross partition boundaries is minimized aiming at low interaction overheads.

On heterogeneous parallel platforms, computation (data) should be partitioned and distributed in proportion to the speed of processors in order to balance the load. Computation performance models are used by application developers to estimate speed of processors. One approach is to build an *analytical* performance model based on static analysis. An analytical performance model provides insights into the performance bottlenecks of parallel applications, and can be used to estimate speed for simple applications, but may not be sufficient for complex applications. A simplistic performance model that has been used extensively in a variety of parallel algorithms sees a heterogeneous network of computers as a set of interconnected processors each is characterized by a single positive constant representing its speed. But in reality, processor speed is application dependent, and it may vary with change of size of the problem to be solved, especially when problem sizes exceed the main memory and cause paging. To represent the performance more accurately, the *functional* performance model has been proposed, which represent the speed of processor as a continuous function of problem size, taking into account both processor heterogeneity and memory heterogeneity [84]. However, currently this type of performance model is only applicable to a heterogeneous network of uniprocessor computers.

Highly heterogeneous multicore platforms, possibly enhanced with hardware accelerators, have become the mainstream in high performance computing. Maximum performance of data-parallel scientific applications on such platforms can be achieved by balancing the workload between heterogeneous processing elements. Data partitioning is a method of static load balancing widely used on distributed-memory platforms, which relies on accurate performance models of processors for generating balanced partitioning of workload. In this thesis, we use data partitioning to balance data-parallel scientific applications. However, on our target platforms, the data partitioning is complicated by several factors, including contention for shared system resources, non-uniform memory access, and so on, which have not been addressed in related works [84, 31]. In this work, we study and provide solutions for the performance modeling and data partitioning for modern heterogeneous parallel platforms [133, 134]. A software framework for facilitating data partitioning on heterogeneous platforms is also presented [34].

1.2 Project Scope

The platforms we target are dedicated highly heterogeneous modern HPC platforms composed of a variety of multicore nodes, and possibly enhanced by hardware accelerator GPUs. The processor cores in a particular multicore node are identical, but multicore processors in a cluster may not be identical due to the different architecture of cores, and/or different on-chip communication mechanisms, and so on, leading to processor heterogeneity. Besides, in a multicore node of NUMA architecture, the performance is dependent on the distance between the processor core and the memory bank where the data are stored. Another property of the target platforms is that the memory capacity of multicore nodes in a cluster may not be identical, leading to memory heterogeneity. Another source of memory heterogeneity is that the GPU memory is significantly smaller than the main memory of the CPU host in hybrid CPU-GPU computing systems.

In this thesis, we target data-parallel scientific applications such as dense linear algebra, digital image processing, computational fluid dynamics. These applications are characterized by divisible computational workload, which is proportional to the size of data. Static distribution methods are usually used for such applications on distributed-memory platforms due to low communication cost and less scheduling overhead. Considerable efforts have been made over recent years to port critical scientific software to both multicore and GPU architectures. This often required re-engineering of the existing parallel applications and development of new programming models, tools and algorithms. Therefore, we do not focus on developing new algorithms or low-level optimization of existing algorithms for target platforms, but reuse the existing highly optimized code for development of scientific software for heterogeneous parallel platforms and apply data partitioning based on performance models to balance the load in order to achieve high performance.

Scientific parallel applications are mainly written using MPI, therefore, we focus on load balancing of MPI parallel implementations on heterogeneous distributed-memory platforms. The integration of multiple processing cores onto the same silicon die has become the de-facto processor design, which make all computers shared memory parallel computers. At this level of parallelism, both MPI implementations and multithreaded OpenMP implementations are considered. For the GPU programming, currently we only experiment with CUDA implementations, but experiments with OpenCL implementations could be included as part of the future work. To sum up, on heterogeneous multicore clusters, both the MPI programming model and the hybrid MPI+OpenMP programming model are taken into consideration. In cases when GPUs are used as hardware accelerators, the hybrid MPI+CUDA programming model is used where the GPUs are handled by host MPI processes running on the dedicated CPU cores.

1.3 Data Partitioning on Modern Heterogeneous HPC Platforms

Heterogeneous multiprocessor systems, where multicore processors are coupled with graphics processing units (GPUs), have been widely used in high performance computing as one approach to continuing performance improvement while managing the new challenge of energy efficiency. Data parallel scientific applications can be load balanced by applying data partitioning with respect to the performance of a heterogeneous platform's computing devices.

Data partitioning algorithm based on functional performance models (FPMs) was originally designed and proved to be accurate for heterogeneous network of uniprocessors. Optimal distribution of computations between heterogeneous processors is typically based on their individual performance models. However, data partitioning on multicore and multi-GPU systems is complicated by several factors, such as contention for shared system resources, non-uniform memory access (NUMA), limited device memory and slow bandwidth of PCIe, which connects the host processor and the device, etc.

In multicore or GPU-accelerated multicore systems, the speed of one processing element may depend on the load of others due to resource contention, therefore, they cannot be considered independent and their speed cannot be measured separately. In this work, we propose to model a multicore or GPU-accelerated multicore system by a set of abstract processors determined by the configuration of the parallel application. Each abstract processor represents a processing unit made of one or a group of processing elements executing a computational kernel of the application. To measure the performance accurately, we propose to group processing units by shared system resources, so that system resources are shared within each group but not between groups. The performance of processing units in a group is measured when all processing units in the group are executing some workload simultaneously, thereby taking into account the influence of resource contention. For example, processing units that share memory or PCIe link are grouped together.

Using the proposed method for measuring performance, the speed of processing units are measured for a number of problem sizes to build their functional performance models. These performance models can be used in FPM-based data partitioning algorithms to balance the workload on heterogeneous multicore platforms or heterogeneous multicore and multi-GPU platforms.

1.4 Domain Decomposition of Computational Fluid Dynamics Applications on Modern Heterogeneous HPC Platforms

Computational Fluid Dynamics (CFD) is the analysis of systems involving fluid flow, heat transfer, and associated phenomena such as chemical reactions by means of computer-based numerical simulation. Over the past few decades, computational fluid dynamics has become a practical cornerstone of most fluid and mechanical engineering applications, such as aerodynamics of aircraft and vehicles.

CFD simulations commonly require a significant amount of computing resources for accurate solutions, especially for complex simulation scenarios such as transonic or turbulent flows. With high performance computing platforms, faster and better solutions can be achieved. Parallelization of numerical simulations of fluid dynamics is an attractive research topic from the very beginning in this area. Parallel computing in CFD simulations is usually based on domain decomposition, which is essentially data parallelism. The current trend in gaining high computing power is to incorporate specialized processing resources such as GPUs in multicore systems, thus making a computing system heterogeneous. Many efforts have been invested in improving the performance of parallel CFD simulations on homogeneous computing systems, but more is required for heterogeneous multicore and multi-GPU platforms.

In this work, using the proposed performance modeling and performance measurement methods on GPU-accelerated multicore systems, we build functional performance models of the platform's processing units executing the linear equation solvers of the CFD application. Then, we apply FPM-based data partitioning to balance the workload of the CFD application on heterogeneous multicore and multi-GPU platforms to speedup the simulation. Experimental results demonstrate that data partitioning algorithms based on proposed methods are able to balance the workload and deliver good performance for such complex applications on target platforms.

1.5 A Software Framework for Data Partitioning on Heterogeneous Platforms

It is challenging to apply data partitioning for parallel scientific applications on heterogeneous platforms. The computation should be partitioned and distributed to heterogeneous processing elements in proportion to their relative speed in order to balance the workload. Therefore, it requires accurate and efficient benchmarking methods to obtain processor speed, appropriate interpolation methods to predict processor performance and build computation performance models, and data partitioning algorithms that can yield high quality partitioning.

To this end, a software framework, Fupermod, has been developed to meet the above challenges. The software framework consists of a library and a set of tools. The library implements the main functionality, which can be integrated into applications. The library is made up of five modules: process configuration, performance measurement, model construction, static data partitioning, dynamic data partitioning and load balancing. The tools, which are developed based on the library, are used in different phases of a model-based data partitioning procedure.

In the software framework, algorithms for benchmarking and constructing computation performance models are implemented. The library provides well-defined function interfaces for invoking these algorithms. A standalone executable tool with the same functionality is also implemented as well. The performance models can be built either in advance to be used in static data partitioning, or at runtime during dynamic data partitioning or dynamic load balancing. A set of general-purpose data partitioning algorithms based on computation performance models are implemented. Similarly, the library releases related function interfaces for invoking these algorithms, and standalone tools used to generate high quality partitioning are also provided. The software framework support a wide range of heterogeneous platforms, such as heterogeneous network uniprocessor or multicore computers, GPU-accelerated hybrid platforms. It is designed to be extensible in that new measurement techniques for new types of hardware can be added and other computation performance models and data partitioning algorithms can be included. The software framework also includes some use cases such as heterogeneous matrix multiplication and Jacobi solver, and some wrappers of CBLAS routines and memory management for heterogeneous processing elements.

1.6 Contributions

The contributions of this thesis are as follows:

- We propose methods of performance modeling of dedicated multicore or GPU-accelerated multicore systems. A multicore or GPU-accelerated multicore system is modeled by a number of abstract processors, each representing a processing unit made of one or a group of processing elements executing a computational kernel of the application. We build functional performance models of abstract processors, and partition data using these performance models.

- We propose methods for accurate performance measurement on dedicated multicore or GPU-accelerated multicore systems. The processing units of a multicore or GPU-accelerated multicore system are grouped by shared resources. Performance of processing units in each group are measured simultaneously rather than separately, thereby taking into account the influence of resource contention.

- We propose a procedure for performance measurement of the parallel MPI application on a heterogeneous multicore cluster. The default communicator is split so that there is one per multicore node. Performance measurement is synchronized on each multicore node. To ensure the reproducibility and reliability of the results, we bind processes to processing elements and repeat experiments multiple times.

- From analysis of functional performance models built in different configurations, we reveal the impact of resource contention on the performance of CPU and GPU processing units, and the impact of process mapping on GPU-accelerated systems of NUMA architecture on the performance of the GPU processing unit.

- We evaluate the proposed performance modeling and performance measurement methods with a fundamental data parallel applications, namely, parallel matrix multiplication. Experimental results demonstrate that data partitioning algorithms based on functional performance models of abstract processors, each representing one or a group of processing elements, are able to balance the workload on heterogeneous multicore or GPU-accelerated multicore platforms.

- We apply the FPM-data partitioning to a complex CFD application, namely numerical simulation of lid-driven cavity flow. Using proposed performance modeling and measurement methods, we built functional performance models of the platform's processing units executing the linear equation solvers of the CFD test case, and decompose the solution domain based on these performance models. Experimental results demonstrate that data partitioning algorithms based on proposed methods are able to balance the workload on heterogeneous multicore or GPU-accelerated multicore platforms.

The technical contributions of this work are as follows:

- We implement the out-of-core single-GPU and multi-GPU *gemm* computational kernels, and investigate a set of optimization techniques.

- We extend the software framework, Fupermod, to heterogeneous multicore clusters and heterogeneous GPU-accelerated multicore platforms, including the code refactoring on process configuration, performance measurement, and the development of wrappers for memory operations and for the execution of linear algebra routines on hybrid GPU-accelerated multicore systems. In addition, during the use of the framework, several bugs have been found and fixed.

1.7 Thesis Structure

The thesis is structured as follows. In Chapter 2, we describe the background and related work, where the existing platforms for high performance heterogeneous computing, the programming models for existing HPC platforms, and existing work on performance modeling and model-based data partitioning on heterogeneous platforms are overviewed. In Chapter 3, we present methods of performance modeling and data partitioning on heterogeneous cluster of multicore nodes, which extends the previous work on heterogeneous uniprocessor clusters. In Chapter 4, we present methods of performance modeling and data partitioning on multicore and multi-GPU platforms, and evaluate the proposed methods with parallel matrix multiplication on both a single hybrid multicore server and a GPU-accelerated multicore cluster. In Chapter 5, we evaluate the proposed methods of performance modeling and performance measurement with a complex computational fluid dynamics application, and analyzed the experimental results. In Chapter 6, we outline the software framework, Fupermod, which is designed to facilitate the performance modeling and data partitioning of parallel scientific applications on heterogeneous platforms. Chapter 7 concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, first, the existing platforms for high performance heterogeneous computing are reviewd. Next, the programming models for developing parallel applications on heterogeneous HPC platforms are overviewed. Then, the general matrix multiplication routine is briefly reviewed. Last, the existing work on performance modeling and data partitioning on heterogeneous platforms is reviewed.

2.1 Modern Heterogeneous HPC Platforms

Microprocessors based on a single processing unit (CPU) drove performance increases and cost reductions in computer applications for more than two decades. However, due to heat dissipation and energy consumption issues, the design of microprocessors has been shifting to a new model where the microprocessor has multiple processing units known as cores [61]. There are two main approaches to such microprocessor design, namely multicore and many-core. The multicore approach integrates a few existing superscalar cores into a single microprocessor, seeking to keep the execution speed of sequential programs. The many-core approach uses a large number of simple smaller cores and is specially oriented to the execution throughput of parallel programs. This approach can be exemplified by the Graphical Processing Unit (GPU) and the Field-Programmable Gate Array (FPGA).

Heterogeneity has become a common attribute of high performance computing platforms. Currently, the dominant type of HPC platforms are computer clusters. However, this type of HPC platforms is naturally heterogeneous. The processors in a cluster may be heterogeneous in terms of architecture, model, and configuration. Moreover, the communication network may have a heterogeneous structure. For example, the computers in a cluster may be connected by network links of different transmission speed.

The demand for heterogeneity in computing systems increases partially due to the need for high performance computing in recent years. The current trend in gaining high computing power is to incorporate specialized processing resources such as GPUs in multicore systems, thus making a computing system heterogeneous [18]. Extensive research has been carried out on general purpose GPU computing [25, 112], and hybrid CPU-GPU computing [19, 101, 100, 24, 88, 59] over recent years. A large number of algorithms and specific applications have been successfully ported to GPUs claiming substantial speedup over their optimized CPU counterparts [94]. In [121], the trends leading to the idea of hybrid CPU-GPU systems is highlighted and a set of techniques that can be used to effectively program such systems are presented in the context of dense linear algebra. In[120], a set of dense linear algebra solvers, namely Cholesky, LU, and QR factorizations, for GPU-accelerated hybrid systems are presented. Large-scale GPU clusters are gaining popularity in the scientific computing community. In [73], the authors present their experiences in deploying two GPU clusters at NCSA, give data on performance and power consumption, and outline solutions for hardware reliability testing, security, job scheduling and resource management.

Several high-level programming systems for heterogeneous GPU-accelerated multicore platforms have been developed. Qilin [88] automatically generates code and uses adaptive mapping for performance tuning. During the training run, Qilin executes the program at different input sizes on CPUs and GPUs separately, and build performance models to determine workload partitioning between CPUs and GPUs. Peppher [14] employs component implementation variants of performance-critical parts of applications tailored to different architectures, and relies on the compiler and runtime system to select and schedule component tasks on available computing resources. PetaBricks [102], an implicitly parallel language and compiler, uses an empirical autotuning approach to search the space of possible implementations at installation time to construct poly-algorithms that combine many different algorithmic techniques to obtain better performance. Other heterogeneous programming systems with similar functionality include Merge [87], StarPU [6], and Elastic Computing [126].

The heterogeneity in modern computing systems gradually goes down to the chip level, as the increase in chip area and the further scaling of fabrication technologies allow for formerly discrete components to be integrated parts of a system-on-chip. For example, the IBM Cell Broadband Engine chooses a heterogeneous chip multiprocessing architecture consisting of one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs) [54, 55]. The PPE delivers system-wide services, such as virtual memory management, threading scheduling and other operation system services. The SPE implements a data-parallel computing architecture based on SIMD RISC computing and explicit data transfer management. The AMD Fusion Accelerated Processing Unit (APU) integrates integrates a few general-purpose x86 processor cores and one integrated programmable vector-processing Graphic core on the same die [17]. The AMD APU aims to strike a balance between performance and power, and optimize performance for different workload classes, e.g. CPU centric and graphic centric. Extensive research has been conducted on such heterogeneous system-on-chip multicore processors in high performance computing community [45, 113, 50, 40, 37, 38], and has proven its promising future in the HPC community.

Heterogeneous architectures, either standard multicore CPU enhanced with specialized resources such as GPUs and FPGAs, or heterogeneous system-on-chip such as IBM Cell and AMD Fusion APU, are of continuous interest in the HPC world. It is challenging and promising, and still requires more effort to fully explore the power of such heterogeneous architectures.

2.2 Parallel Programming Models

Just as there are several different classes of parallel hardware, so too are there several distinct models of parallel programming. Currently, the OpenMP [99] is the most widely accepted parallel programming model for shared-memory systems, and the MPI [92] is the dominant programming model for distributed-memory systems. Modern HPC platforms usually employ both shared- and distributed-memory architectures. For example, a multicore cluster consists of a number of shared-memory multicore nodes connected by network links. Combining the OpenMP and MPI offers an approach to exploit the hierarchical parallelism inherent in the applications or the underlying hardware [23]. Considerable work has gone into studying this hybrid programming model [105, 106, 107, 114, 123].

In the hybrid MPI+OpenMP programming model, one may invoke one or more MPI processes on a multicore node, each further invoking multiple threads on processor cores using OpenMP. To get good performance, it is important to take into account the hardware topology, intra-node MPI communication, and network saturation [106]. Optimal thread-core affinity could lead to less inter-node communication which impacts the application performance. Depending on the inter-node, intra-node, and intra-socket bandwidths, rank orderings should be chosen accordingly. The number of MPI processes or OpenMP threads for communication should be chosen carefully in order to saturate the network connection of a node.

Modern HPC platforms present high heterogeneity due to the additional specialized resources such as GPUs. In the early stage, languages such as Brook [19] and Cg [89] were used for programming on the GPU-accelerated computing systems. At present, CUDA [97] is the widely used programming model for NVIDIA GPUs. The CUDA programming model assumes that the computing system consists of a *host* (CPU) and one or multiple *devices* (GPU), which maintain their own memory spaces respectively. In this programming model, the programmer needs to allocate memory on the device and transfer the input data from the host memory to the allocated device memory in order to execute a kernel on a device. After device execution, the programmer needs to transfer the result from the device memory back to the host memory and free the device memory that is no longer needed. The CUDA runtime system provides application programming interface (API) functions to perform these activities. More recently, the industry has worked together on the OpenCL standard [72] for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. OpenCL is designed to target not only GPUs but also other accelerators, such as multi-core CPUs. Thus, it can support both data-based parallelism, and task-based parallelism, which are well suited for GPUs and CPUs architectures respectively.

For hybrid CPU+GPU platforms, parallel programming models for CPUs such as OpenMP and MPI, and programming models for GPUs such as CUDA and OpenCL can be used together to explore the computing power of such hybrid platforms [16, 41, 56, 73, 108, 130]. For example, in [65], OpenMP has been used to generate as much data as possible (data preprocessing) to feed the GPU. In [56], OpenMP has been used to support multi-GPU parallelism. That is, multiple CPU worker threads are allocated by OpenMP instructions, and each thread manages one GPU. In [41], the author used a

OpenMP-CUDA solution that runs on a loosely-coupled GPU cluster to triangulate and render large molecular datasets with a significantly large number of atoms.

To parallelize programs in GPU clusters, a natural method is to combine CUDA with MPI. This hybrid programming model is becoming an important choice for various HPC applications where MPI works as the data distributing mechanism and the CUDA/GPU as the local computing engine [16, 108]. Most hybrid CUDA+MPI code is written in a hierarchical way. Namely, at a high level, the program is structured as several MPI tasks to execute on nodes of the GPU cluster. In each MPI task, the parts which enrich data parallelism are moved to the GPU to execute. To get good performance on multi-GPU nodes of NUMA architecture, it is important to take into account the impact of process affinity on the application performance, because GPUs have different memory bandwidth to the host CPU cores depending on what CPU socket the GPU control process is running on [73].

2.3 A Brief Review of the General Matrix Multiply Routine

General matrix-matrix multiply (*gemm*) is a fundamental linear algebra routine from the level-3 Basic Linear Algebra Subprograms (BLAS) [43]. High performance of other level-3 BLAS routines, such as symmetric matrix-matrix multiply (*symm*), symmetric rank-k update (*syrk*), and triangular matrix-matrix multiply (*trmm*), are generally attained by casting the bulk of computation in terms of a general matrix-matrix multiply (*gemm*) [68]. The BLAS subroutines have been successfully used as building blocks for several higher-level math programming languages and libraries, including LAPACK [4] and ScaLAPACK [28].

Highly optimized BLAS implementations, each including a optimized *gemm* routine, have been developed by hardware vendors, such as Intel MKL [63] and AMD ACML [1] libraries, and by other researchers, such as GotoBLAS [52] and ATLAS [127] libraries. In the optimized *gemm* implementations, the matrices are generally partitioned into smaller blocks, taking into account the size of caches and bandwidth between different memory layers of the computing system. Then the general *gemm* is decomposed into multiple calls to some basic kernels, such as general block-panel multiply (*gebp*), general panel-block multiply (*gepb*), and general panel-panel product (*gepdot*), which operate on the small matrix blocks stored in the caches. The *gemm* routine is designed so that the required data movements between memory layers are optimally amortized [52, 57]. Generally, the *gemm* routine should be tuned for different architectures. In [127], an approach for automatic generation and optimization of the BLAS routines for computing systems with deep memory hierarchies is proposed.

With the advert of multicore and GPU architectures, *gemm* routines optimized for such architectures have been proposed. In [110], the matrix multiplication is recursively decomposed into a large number of smaller matrix multiplication tasks. These tasks are then mapped and scheduled to cores by a task administration library, and are executed by multiple threads in parallel. Multithreaded versions of the *gemm* routine are also available from hardware vendors such as the Intel and AMD. They can be used straightforwardly on multicore systems. Extensive work has been conducted on developing high performance *gemm* routines for GPUs. In [124], a set of benchmarks are designed, which are used to reveal the bottlenecks and structure of the GPU. Based on benchmarks, a high performance hand-tuned *gemm* routine is developed. In [93], the above algorithm is improved for more advanced NVIDIA Fermi GPUs with extended memory hierarchy and memory sizes. In [75], an approach for auto-tunning the *gemm* routine for NVIDIA Fermi GPUs is proposed. Vendor-optimized BLAS implementations including the *gemm* routines are also available, such as the clBLAS library [2] designed for AMD GPUs and the cuBLAS library [95] designed for NVIDIA GPUs.

2.4 Data Partitioning on Heterogeneous Platforms

Dividing a computation into smaller computations, and assigning them to different processors for parallel execution are the two key steps in the design of parallel algorithms [74]. Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps. In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks.

Once a computation has been decomposed into tasks, these tasks are mapped onto processes with the objective that all tasks complete in the shortest amount of elapsed time. In order to achieve a small execution time, the overheads of executing the tasks in parallel should be minimized. For a given decomposition, the two major sources of overhead are *load imbalance* and *inter-process communication*. These two objectives often conflict with each other, and finding a good mapping is a nontrivial problem. Mapping techniques used in parallel algorithms can be broadly classified into two categories: *static* and *dynamic*.

Static mapping techniques, such as those based on data partitioning [98, 84, 88], distribute the tasks among processes prior to the execution of the algorithm. Static algorithms are particularly useful for applications where data locality is important because they do not require data redistribution. However, these algorithms are unable to balance on non-dedicated platforms, where load changes with time, and for applications with non-deterministic workload. Dynamic mapping techniques, such as task queue scheduling and work stealing [62, 87, 6, 7, 103, 104], distribute the work among processes during the execution of the algorithm. While algorithms that make use of static mapping are in general easier to design and program, algorithms that require dynamic mapping are usually more complicated.

Dynamic mapping techniques do not require a priori information about execution but may incur significant communication overhead on distributed memory platforms due to data migration. Dynamic algorithms often use the static data partitioning for their initial step to minimize the amount of data redistributions needed. For example, in the state-of-the-art load balancing techniques for multi-node, multicore, and multi-GPU platforms, the performance gain is mainly due to better initial data partitioning. It was shown that the static distribution based on a simplistic performance model improves the performance of traditional dynamic scheduling techniques by up to 250% [115].

2.4.1 Performance Models of Processors

The performance models of processors are important for solving data partitioning problems. One approach to prediction is to build an *analytical* performance model based on static analysis. Indeed, such models [66, 128, 60, 8, 132] are useful to provide insights into the performance bottlenecks of parallel applications on CPU or GPU architectures, and can be used to estimate the execution time of applications for simple applications. However, it is unlikely to be sufficient for more complex applications.

Constant Performance Model

Heterogeneity is one of the main sources of performance programming issues [131]. 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.

The performance of a processor can be calculated theoretically or measured by benchmark suites. The peak performance provided by manufacturer is perhaps the simplest measure of the processor performance. While the peak performance is theoretical and rarely achieved by real applications due to on-chip bandwidth and latency, it can be used as initial values in adaptive data partitioning algorithms [129]. As an alternative to theoretical estimations, benchmarks may be executed to characterize the performance of a processor, such as SPEC [49] for general-purpose CPUs and Parboil [117] for GPU architectures. As heterogeneous computing systems consisting of both multicore and accelerators (e.g., GPU, FPGA) become increasingly popular, benchmark suites targeting such systems are developed, like Rodinia [24] and SHOC [39].

In data partitioning the typical representations of performance are processor weight, relative speed, or normalized speed of the set of processors, which are essentially the same and assume the performance is constant for all problem sizes. Based on the relative speed, the workload is proportionally divided between processors so that the computation of their respective work will complete at the same time as expected. Often data partitioning algorithms assume that the relative speed is provided [10, 36, 67, 44]. Otherwise, users are required to use self-defined benchmark routines to calculate the relative speed for optimal distribution of computation between processing elements. In [88], a database that provides execution-time projection for all programs it has ever executed is maintained. In [129], a similar framework is implemented, but the performance is obtained adaptively using a history of performance measurements.

The constant performance model (CPM) assumes that the relative speed of heterogeneous processors is constant and does not depend on the size of the computational task solved by the processors. In reality, the relative speed changes due to processor heterogeneity and memory heterogeneity [84]. Thus, the constant performance model is not accurate and realistic. As processor architectures become more heterogeneous, their performance as problem size increases becomes more difficult to approximate and this necessitates a more detailed general model of performance than constant performance model. In order to address these issues, the Functional Performance Model (FPM) has been proposed in [84] which represents the performance of a processor as a continuous function of problem size. It is built empirically and integrates many important features characterizing the performance of both the architecture and the application.

Functional Performance Model

Under the functional performance model (FPM) [84], the speed of each process is represented by a continuous function of problem size. The speed is defined as the number of computation units performed by the process per one time unit.

The *computation unit* is defined as the smallest amount of work that can be given to a process. The model is application specific. In particular, this means that the computation unit can be defined differently for different applications. The important requirement is that the computation unit must not vary during the execution of the application. All units require the exact same number of arithmetic calculations and have the same data storage requirements. An arithmetical operation and the matrix update $C = C + A \times B$, where A, B, and C are $r \times r$ matrices of the fixed size r, give us examples of computation units.

The *problem size* is understood as a set of one, two or more parameters characterizing the amount and layout of data stored and processed during the execution of the computational task. The number and the semantics of the problem size are problem- or even application-specific. It is assumed that the amount of stored data will increase with the increase of any of the problem size parameters.

Some assumptions are made about the shape of the function. Namely, it is assumed that along each of the task size variables, either the function is monotonically decreasing or there exists point x such that on the interval $[0,x]$ the function is monotonically increasing, concave, and any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point, and on the interval $[x,\infty)$ the function in monotonically decreasing. Figure 2.1 shows the functional performance models of heterogeneous processors from Grid5000. As we can see, the relative speed can be seen constant only in a small range of problem sizes [29].

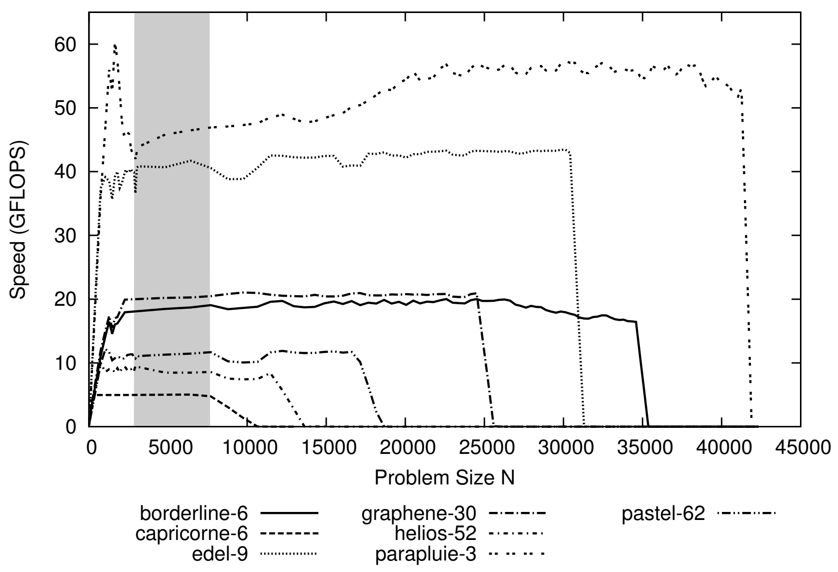


Figure 2.1: Speeds of multiplication of two square $N \times N$ matrices observed on heterogeneous processors of Grid5000 [29]. The shaded area indicates the range of problem sizes where the relative speed can be seen constant.

Data partitioning on heterogeneous platforms requires detailed knowledge of the performance of the specific applications executed on targeted platforms. The functional performance model is a general solution to providing this performance information. In [83, 58] the construction of the functional performance model has been described. Application of such performance models in various scenarios has resulted in increased performance when using heterogeneous resources. This has been demonstrated for data partitioning [78, 79, 84, 80, 82, 81, 32, 33, 31, 34] and task scheduling [58].

2.4.2 Data Partitioning Based on Performance Models

In this section, we describe data partitioning algorithms based on the constant performance model and functional performance model respectively.

CPM-based Data Partitioning

Under the constant performance model, each processor in a heterogeneous platform is represented by a positive constant. 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 can be either *absolute* or *relative*. The absolute speed of a processor is understood as the number of computation units performed by the processor per one time unit. The relative speed can be obtained by the normalization of its absolute speed so that $\sum_{i=1}^{p} = 1$.

In a general form, a typical partitioning problem with a constant performance model of heterogeneous processors can be formulated as follows [77, 76]. Given a set of p processors $P_1, P_2, \ldots, P_p$, the speed of each processor is characterized by a positive constant s_i. Partition a mathematical object (e.g., set, matrix, graph) of the size n (the number of element in a set or matrix, or nodes in a graph) into p subobjects (e.g., subset, submatrix, subgraph) of the same type so that (1) there is one-to-one mapping between the partitions and the processors; (2) the size n_i of each partition is approximately proportional to the speed of the processor owning the partition, $n_i/s_i \approx const$, assuming the volume of computation is proportional to the size of the mathematical object, and the notion of proportionality is supposed to be defined for each particular problem. (3) the partitioning may need to satisfy some additional restrictions on the relationship between the partitions or minimize some functionals used to estimate each partitioning.

CPM-based data partitioning has been used in many applications. In [11], the matrix is partitioned into a two-dimensional grid based on a constant performance model so that the workload is balanced and the total volume of communication is minimized. [70] provides another solution to optimal partitioning of matrix between heterogeneous processors without taking communication overhead between processors into account. In [80], the problem of LU factorization of a dense matrix on a heterogeneous platform is reduced to the problem of partitioning a well-ordered set with a constant performance model of a heterogeneous platform. In [86, 47, 90], the speeds of processors in heterogeneous platforms are represented by positive constants for dynamic load balancing of iterative computational applications with the constant performance model updated iteratively. In [12, 13], parallel algorithms using nonrectangular partitioning based on constant performance models can outperform their counterparts based on the rectangular one, if the number of heterogeneous processors is small.

FPM-based Data Partitioning

The problem of data partitioning using functional performance models was formulated in [84] as follows. A total problem size n is given as the number of computation units to be distributed between p ($p \ll n$) processes $P_1, \dots, P_p$. The speeds of processors are represented by positive continuous functions of problem size $s_1(x), \dots, s_p(x) : s_i(x) = x/t_i(x)$, where $t_i(x)$ is the execution time of processing x units on the processor i. Speed functions are defined at $[0,n]$. The output of the algorithm is a distribution of computation units, $d_1, \dots, d_p$, so that $d_1 + d_2 + \dots + d_p = n$. Load balancing is achieved when all processors complete their work at the same time: $t_1(d_1) \approx t_2(d_2) \approx \dots \approx t_p(d_p)$. This can be expressed as:

$$\begin{cases} \dfrac{d_1}{s_1(d_1)} \approx \dfrac{d_2}{s_2(d_2)} \approx \dots \approx \dfrac{d_p}{s_p(d_p)} \\ \\ d_1 + d_2 + \dots + d_p = n \end{cases} \tag{2.1}$$

The solution of these equations can be represented geometrically by intersection of the speed functions with a line passing through the origin of the coordinate system, as shown in Figure 2.2 for $p = 4$.

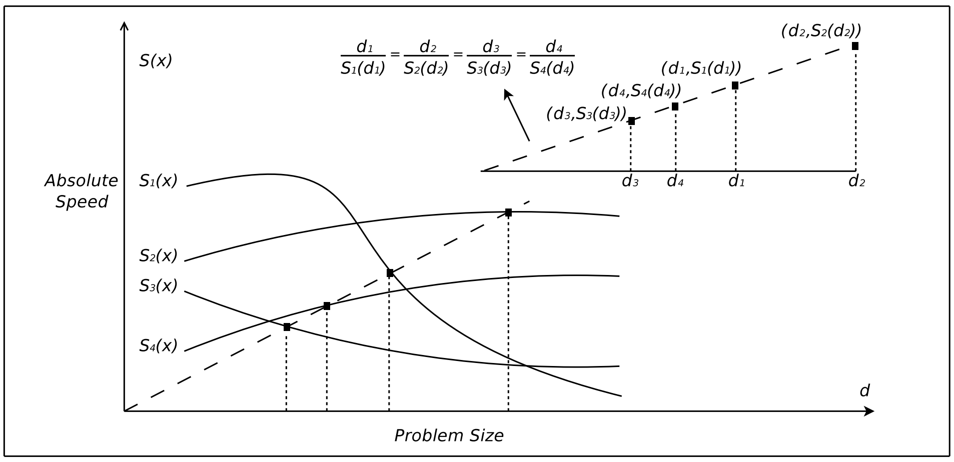


Figure 2.2: Optimal data distribution showing the geometric proportionality of the number of chunks to the speed of the processor [84]

The geometrical algorithm solving this data partitioning problem, as illustrated in

Figure 2.3, was proposed in [84] and can be summarized as follows. Any line passing through the origin and intersecting the speed functions represents an optimum distribution for a particular problem size. Therefore, the space of solutions of the data partitioning problem consists of all such lines. The two outer bounds of the solution space are selected as the starting point of algorithm. The upper line, U, represents the optimal data distribution $d_1^u, \ldots, d_p^u$ for some problem size $n_u < n$, $n_u = d_1^u + \ldots + d_p^u$, while the lower line, L, gives the solution $d_1^l, \ldots, d_p^l$ for $n_l > n$, $n_l = d_1^l + \ldots + d_p^l$. The region between two lines is iteratively bisected by new lines B_k. At the iteration k, the problem size corresponding to the new line intersecting the speed functions at the points $d_1^k, \ldots, d_p^k$ is calculated as $n_k = d_1^k + \ldots + d_p^k$. Depending on whether n_k is less than or greater than n, this line becomes a new upper or lower bound. Making n_k close to n, this algorithm finds the optimal partition of the given problem $d_1, \ldots, d_p$: $d_1 + \ldots + d_p = n$. Correctness proof and complexity analysis of this algorithm are presented in [84].

Functional performance models are built empirically by benchmarking the kernel for a range of problem sizes. The accuracy of the model depends on the number of experimental points used to build it. Despite the kernel being lightweight, building the

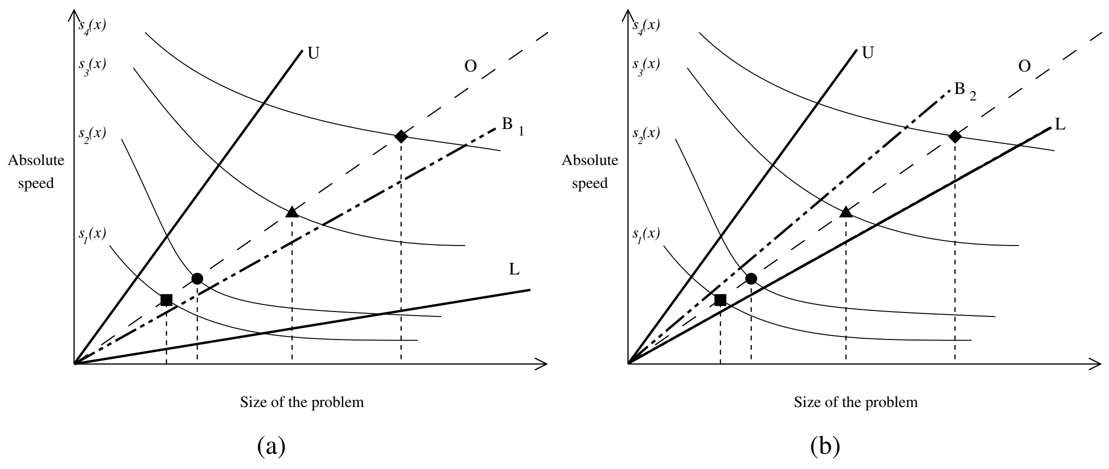


Figure 2.3: Two steps of the iterative geometrical data partitioning algorithm. The dashed line O represents the optimal solution. (a) Upper line U and lower line L represent the two initial outer bounds of the solution space. Line (B_1) represents the first bisection. (b) Line B_1 becomes line L. Solution space is again bisected by line B_2, which, in the next step will become line U. Through this method the partitioner converges on the optimal solution [84].

full model can be very expensive. The applicability of FPMs built for the full range of problem sizes is limited to parallel applications executed many times on stable in time heterogeneous platforms. In this case, the time of construction of the full FPMs can become very small compared to the accumulated performance gains during the multiple executions of the optimized application. However, this approach is not suitable for applications that will be run a small number of times on a given platform, for example, in grid environments, where different processors are assigned for different runs of the application. Such applications should be able to optimally distribute computations between the processors of the executing platform assuming that this platform is different and *a priori* unknown for each run of the application.

Partial estimates of the full speed functions can be built dynamically at application run-time to a sufficient level of accuracy to achieve load balancing [81, 32, 30]. We refer to these approximations as partial functional performance models. The partial FPMs are based on a few points connected by linear segments and estimate the real functions in detail only in the relevant regions: $\bar{s}_i(x) \approx s_i(x)$, $1 \le i \le p$, $\forall x \in [a,b]$. Both the partial models and the regions are determined at runtime.

The algorithm to build the partial FPMs is iterative and alternates between (i) benchmarking the kernel on each process for a given distribution of workload and (ii) repartitioning the data. At each iteration, the current distribution $d_1,\ldots,d_p$ is updated, converging to the optimum, while the partial models $\bar{s}_1(x),\ldots,\bar{s}_p(x)$ become more detailed. Initially the workload is distributed evenly between all processes. Then the algorithm iterates as follows:

1. The time to execute the kernel for the current distribution is measured on each process. If the difference between timings is less than some ε, the current distribution solves the load balancing problem and the algorithm stops.

2. The speeds are calculated from the execution times and the points (d_i, s_i) are added to the corresponding partial models $\bar{s}_i(x)$. (Figure 2.4 (b, d, f)).

3. Using on the current partial estimates of the speed functions, the FPM-based partitioning algorithm calculates a new distribution. (Figure 2.4 (a, c, e)).

This algorithm allows for efficient load balancing and suitable for use in self-adaptable applications, which run without *a priori* information about the heterogeneous platform.

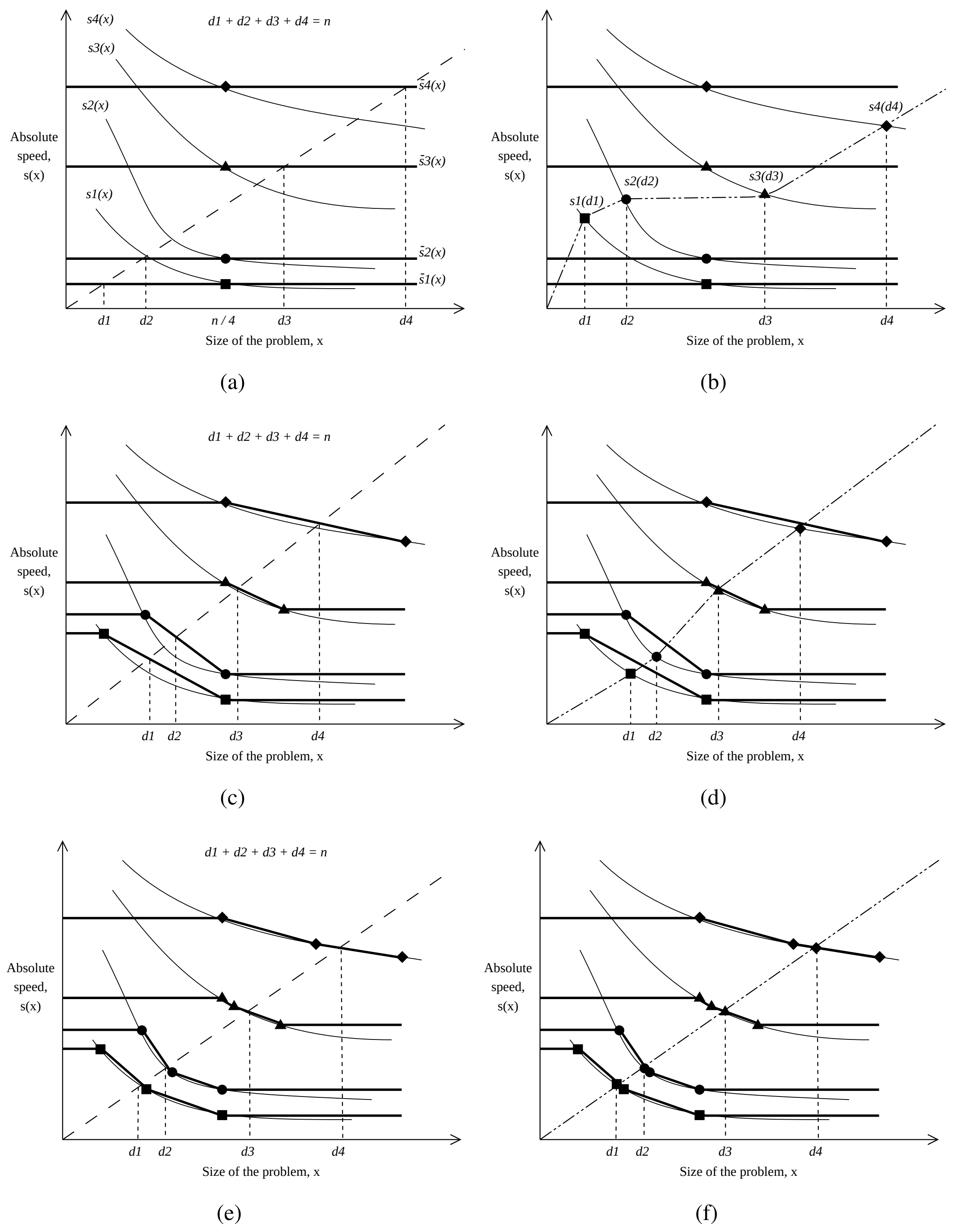


Figure 2.4: Steps of the partial FPM-based data partitioning algorithm illustrated using four heterogeneous processors [81].

Chapter 3

Data Partitioning on Heterogeneous Multicore Platforms

Data partitioning is a method of static load balancing widely used on distributed-memory platforms. Data partitioning algorithms rely on accurate performance models of processors. One approach is to represent the processor speed by a positive constant, which is known as constant performance model. The fundamental assumptions of data partitioning algorithms based on constant performance models are that (i) the absolute speed of processing elements does not depend on the size of a computational task, and therefore, can be represented by a constant; (ii) the processing elements are independent of each other, and thus, their speed can be measured separately. However, these assumptions become invalid in the following situations:

1. the partitioning of the problem results in tasks fitting into different levels of memory hierarchy;

2. processing elements switch between different codes to solve the same computational problem;

3. processing elements contend for shared system resources with each other.

Data partitioning algorithms based on functional performance models (FPMs) were originally designed and proved to be accurate for heterogeneous network of uniprocessors [84]. The functional performance model represents the processor speed by a function of problem size. It is built empirically, and integrates many important features characterizing the performance of both the architecture and the application. Nevertheless, this performance model is only applicable in situations (1) and (2).

In this chapter, we focus on the optimal data distribution of data parallel scientific applications on heterogeneous multicore platforms. Data parallel applications are characterized by divisible computational workload, such as dense linear algebra and computational fluid dynamics. The computational workload is proportional to the size of data. Maximum performance of data parallel scientific applications on a heterogeneous multicore platforms can be achieved by balancing the load between heterogeneous processing elements. In order to distribute workload between heterogeneous processing elements optimally, we use data partitioning algorithms. On multicore systems, the speed of one CPU core may depend on the load of others due to resource contention (situation 3), therefore, they cannot be considered independently and their speed cannot be measured separately. We propose methods of performance modeling and performance measurement on multicore systems, and extend the FPM-based data partitioning algorithms to heterogeneous clusters of multicore computers.

3.1 Performance Modeling of Multicore Systems

In this work, we propose to model a multicore system by a number of abstract processors. Each abstract processor represents a processing unit made of one or a group of CPU cores executing one computational kernel of the parallel application. In scientific applications, both single- and multi-threaded computational kernels are commonly used. If a single-threaded computational kernel is used, on each CPU core one process executes a computational kernel, then each CPU core will be modeled by an abstract processor. If a multi-threaded kernel is available, one master process executes a computational kernel on multiple CPU cores with multiple threads. Then each group of CPU cores executing a computational kernel make a combined processing unit, and will be modeled as an abstract processor. The master process is supposed to handle the placement and execution of threads, and workload distribution between threads.

On a multicore platform, a parallel application allows for various configurations, depending on the computational kernel used in the application and their mapping to CPU cores. For example, on a multicore node, if multi-threaded computational kernel is used, multiple kernels could be executed with each on a group of CPU cores of different number, leading to heterogeneous processing units. Nevertheless, in this work, we assume a parallel application will be configured following a set of simple rules, which we believe lead to efficient configurations: (i) all CPU cores of each multicore node will be utilized. (ii) if a multi-threaded computational kernel is used, each group of CPU cores that share resources will only execute a single such computational kernel. Thus, each group of CPU cores that share resources makes a combined processing unit, and will be represented by one abstract processor. In this way, CPU processing units that share system resources will always be identical.

The performance of CPU processing units are measured for a wide range of problem sizes to build their functional performance models, which will be used as the input of FPM-based data partitioning algorithms. To measure the performance of processing units accurately, we propose to group processing units by shared resources so that the resources are shared within each group but not shared between groups. The performance of processing units in a group are measured when all processing units in the group are executing some workload simultaneously, thereby taking the influence of resource contention into account. Since no evidence has been found in experiment or in literature to prove that uneven distribution of workload to identical processing units could speed up parallel applications, processing units that share system resources are assigned the same amount of workload during performance measurement. This simplification reduces the complexity of data partitioning problem. Performance measurements of the processing units that share system resources are synchronized. With the same amount of workload, measurements will be completed with roughly the same amount of elapsed time, which realistically simulates resource contention. Performance models of processing units that are built separately do not reflect their actual performance during the execution of the application. Consequently, any load balancing decision based on these performance models will not be accurate.

Figure 3.1 shows a multicore server of NUMA architecture executing a parallel application in two different configurations. In Figure 4.1(a), one process executes one single-threaded computational kernel on each CPU core. Therefore, each CPU core is modeled by an abstract processor representing a CPU processing unit made of one core. As the CPU cores of a NUMA node are tightly coupled and share memory, they cannot be considered independent. To measure the performance accurately, the CPU cores of each NUMA node are grouped together. The performance of the CPU cores

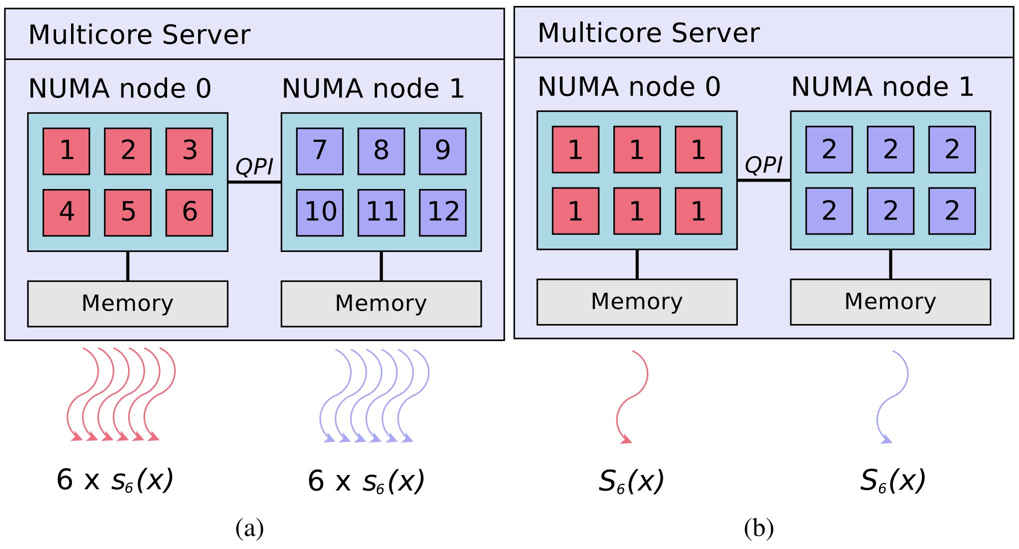


Figure 3.1: Performance modeling on a multicore server of NUMA architecture with (a) a single-threaded kernel executed on each core; (b) a multi-threaded kernel executed on each NUMA node, one thread per core

of a NUMA node are measured simultaneously. As NUMA nodes are relatively independent, the performance of the CPU cores of different NUMA nodes can be measured separately. In Figure 4.1(b), one process executes one multi-threaded computational kernel on six cores of each NUMA node with multiple threads. Therefore, each NUMA node is modeled by an abstract processor representing a combined processing unit made of six CPU cores, and their performance can be measured separately.

In order to build the functional performance models, the performance of the CPU processing units is measured for a wide range of problem sizes. During performance measurement, to prevent the operating system from migrating processes excessively, processes are bound to CPU cores. Processes are synchronized to minimize the idle computational cycles, aiming at the highest floating point rate for the application. Synchronization also ensures that the resources will be shared between the maximum number of processes, generating the highest memory traffic. To ensure the reliability of measurements, measurements are repeated multiple times, and the average execution times are used. We find the confidence interval and stop the measurements if the sample mean lies in the interval with the confidence level 95%. In this work, for simplicity, we use Student's *t*-test, assuming that the individual observations are independent and their population follows the normal distribution.

Three types of functional performance models for CPU cores are defined as follows:

1. $s(x)$ approximates the speed of a uniprocessor executing a single-threaded computational kernel. The speed $s(x) = x/t$, where x is the number of computation units, and t is the execution time.

2. $s_c(x)$ approximates the speed of one of c CPU cores all executing the same single-threaded computational kernel simultaneously. The speed $s_c(x) = x/t$, where x is the number of computation units executed by each CPU core, and t is the execution time.

3. $S_c(x)$ approximates the collective speed of c CPU cores executing a multi-threaded computational kernel. The speed $S_c(x) = x/t$, where x is the total number of computation units executed by all c CPU cores, and t is the execution time. $S_c(cx)/c$ is used to denote the average speed of a CPU core.

Figure 3.2 shows speed functions of a CPU core built in different configurations on Pluto, specified in Table 3.1. Pluto consists of eight identical NUMA nodes, with 6 cores and 16 GB local memory each. Speed functions $s_1(x)$, $s_6(x)$, and $s_{12}(x)$ are built by executing a single-threaded ACML *gemm* kernel per CPU core on only one CPU core, on six CPU cores of a NUMA node, and on twelve CPU cores of two NUMA nodes respectively. Speed functions $S_6(6x)/6$ and $S_{12}(12x)/12$ approximate the average speed of a CPU core when executing a multi-threaded ACML *gemm* kernel on one and two NUMA nodes respectively. We can see that $s_6(x)$ is clearly lower than $s_1(x)$, which indicates extensive resource contention between CPU cores of the same NUMA node. At the same time, there is no difference between $s_6(x)$ and $s_{12}(x)$, which indicates no contention between CPU cores of different NUMA nodes. Therefore, the performance model of the CPU cores of a NUMA node can be built separately from other CPUs. $S_{12}(12x)/12$ is lower than $S_6(6x)/6$ due to inappropriate use of the multi-threaded kernel, which is NUMA-unaware by its design.

Table 3.1: Specifications of the Pluto server *pluto.icl.utk.edu*

Architecture	Core Clock	Number of Cores	Memory Size
AMD Opteron 6172	2.1 GHz	8×6 cores	8×16 GB

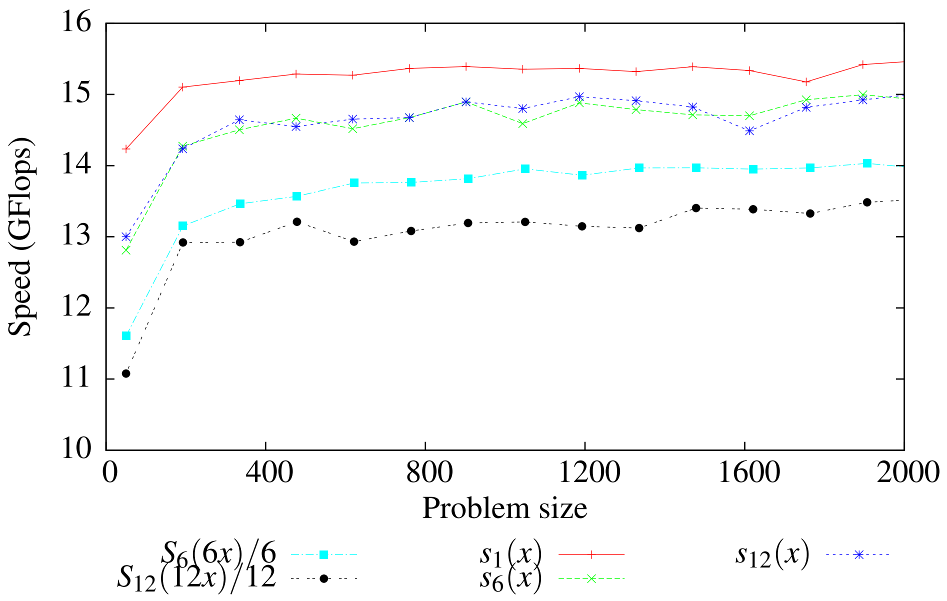


Figure 3.2: Speed functions of a CPU core built in different configurations

Thus we can conclude that depending on the configuration of the application the speed of individual CPU cores can vary significantly. Therefore, to achieve optimal distribution of computations it is very important to build and use speed functions which accurately reflect the performance of CPU cores during the execution of the application.

3.2 Performance Measurement on Multicore Platforms

To build functional performance models of multicore nodes of a heterogeneous multicore cluster, we design a benchmarking procedure that allows us to measure the execution time accurately on multicore nodes simultaneously. For simplicity, we present the benchmarking procedure for parallel MPI applications configured to run with one process per CPU core, as illustrated in Figure 3.3.

Existing performance benchmarks can be categorized into two groups: synthetic benchmarks that measure the sustainable memory bandwidth considering that most scientific applications are memory-intensive and insufficient memory bandwidth is the main factor affecting the performance [91]; floating-point kernels of typical numerical methods that are very important for computational science and engineering[15]. In this work, the computational kernel of the parallel application, which includes contributions from both arithmetic and memory operations, is used for performance benchmark.

During the performance benchmark, each process is bound to a CPU core. This can improve performance if the operating system is placing processes suboptimally. For example, it might oversubscribe some multi-core processor sockets, leaving other sockets idle; this can lead processes to contend unnecessarily for common resources. Process binding can also keep the operating system from migrating processes excessively, regardless of how optimally those processes were placed to begin with. Automatic and explicit techniques for memory affinity management improve access to shared memory, especially on multicore systems of NUMA architecture. In this work, we do not use any explicit memory management techniques but rely on operating systems and MPI implementations.

The benchmarking procedure is summarized as follows:

1. The default communicator that contains all processes is split into multiple intra-node MPI communicators so that there is one intra-node communicator per multi-core node. The workload is partitioned evenly between nodes; then, the workload assigned to each node is further partitioned evenly between CPU cores.

2. A barrier synchronizes the processes within each node.

3. The execution time of the routine is measured on each CPU core.

4. Statistical analysis of all time measurements observed so far for the given problem size is performed.

5. If all intra-node processes get statistically reliable results, the speed of CPU cores is calculated. Otherwise, more repetitions are repeated (`GOTO 2`).

In the benchmarking procedure, one MPI process is executed on each CPU core, therefore, there is a one-to-one mapping between CPU cores and abstract processors. Performance measurements are synchronized on each multicore node. With the same amount of workload, measurements will be completed with roughly the same amount of elapsed time, which realistically simulates resource contention. To ensure the reliability of measurements, measurements are repeated multiple times, and the average execution times

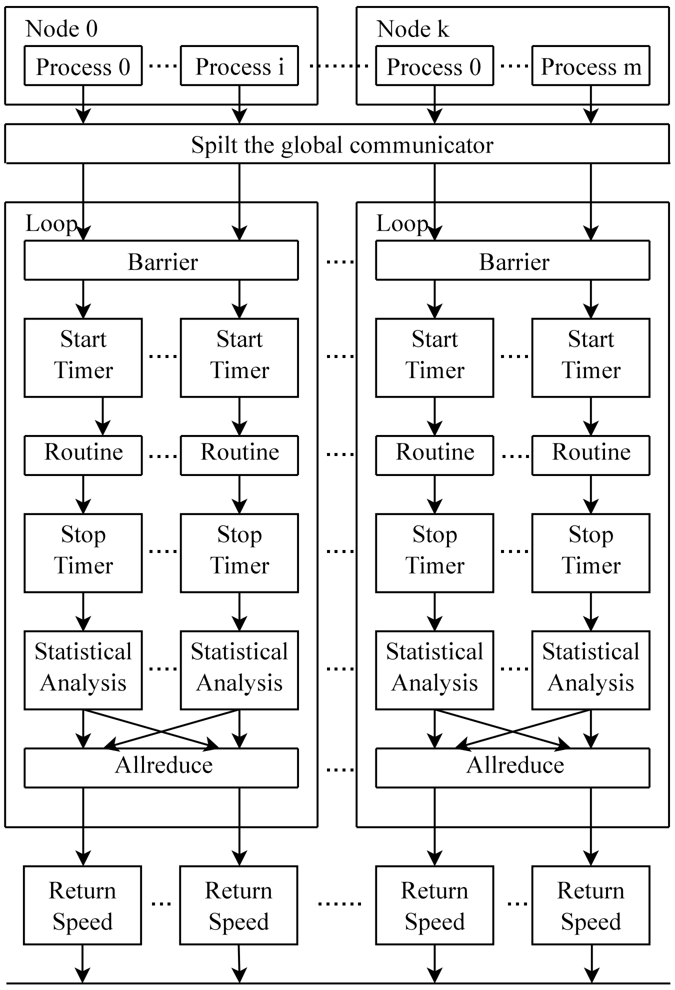
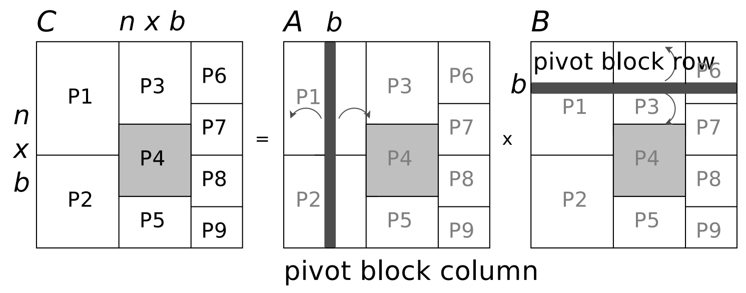


Figure 3.3: Benchmarking procedure on a heterogeneous cluster of multicores

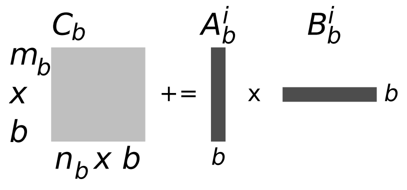
are used. We find the confidence interval and stop the measurements if the sample mean lies in the interval with the confidence level 95%.

3.3 Heterogeneous Parallel Matrix Multiplication

In this section, we present a fundamental data-parallel application that will be used in experiments, namely, the column-based heterogeneous modification [33] of the two-dimensional blocked matrix multiplication [27]. It was originally designed for heterogeneous network of uniprocessors. It takes the functional performance models of heterogeneous processors as input, partitions the matrices using the FPM-based data partitioning



(a)



(b)

Figure 3.4: Heterogeneous parallel column-based matrix multiplication. (a) One step of the algorithm. (b) Computational kernel.

algorithm, and then performs the blocked matrix multiplication using vendor-optimized *gemm* kernels.

Parallelism in this application is achieved by slicing the matrices, with a one-to-one mapping between slices and processes. For efficiency and scalability, the application uses two-dimensional slicing of the matrices. The general solution for finding the optimum matrix partitioning for a set of heterogeneous processors has been shown to be NP-complete [11]. By applying a column-based matrix partitioning restriction, an algorithm with polynomial complexity can be used to find optimum partitioning [69]. In this algorithm each process is responsible for calculations associated with a rectangular submatrix. These rectangles are arranged into columns and the area of the rectangles is proportional to the speed of the processing unit upon which the process is running, as shown in Figure 3.4(a). A communication minimizing algorithm proposed in [11] uses this column-based partitioning and finds the shape and ordering of these rectangles such that the total volume of communication for parallel matrix multiplication is minimized.

In this application, matrices *A*, *B* and *C* are partitioned over a two dimensional arrangement of heterogeneous processes so that the area of each rectangle is proportional to the speed of the process that handles the rectangle. Without loss of generality we work with square $N \times N$ matrices and we assume that N is a multiple of the blocking factor b. Dense matrices A, B and C are partitioned into p submatrices where p is the number of processes. The application consists of $n = N/b$ iterations. At each iteration pivot column of matrix A and pivot row of matrix B are broadcast horizontally and vertically respectively. The pivot column and row move horizontally and vertically respectively with each iteration. Figure 3.4(a) shows one iteration. If a process owns part of the pivot column, then it will broadcast its part of the pivot column horizontally. If a process owns part of the pivot row, then it will broadcast its part of the pivot column vertically. At each iteration, each process will receive its parts of the pivot column and pivot row into a buffers A_b and B_b, and update the rectangle it handles, C_b, by performing *gemm* operation. So the whole matrix C is updated in parallel by all processes updating their own parts. The partitioning algorithm used in this application arranges the submatrices to be as square as possible in order to minimize the total volume of communications and balancing the computations on the heterogeneous platform [33]. The blocking factor b is a parameter of the application and used to adjust the granularity of communications and computations [27]. The optimal value of this parameter depends on the architecture of platform and the implementation of the *gemm* routine, and is found experimentally.

The absolute speed of the processor is defined as the total number of computations executed by this processor during the application divided by the total execution time. In order to measure the speed more efficiently, we make an assumption that the total execution time of the application can be approximated by multiplying the number of iterations of the application, n, with the execution time of a single run of the computational kernel. As shown in Figure 3.4(b), the computational kernel performs one update of the submatrix C_b with the portions of pivot column A_b^i and pivot row B_b^i: $C_b += A_b^i \times B_b^i$. Therefore, this speed is estimated more efficiently by measuring just one run of the kernel. This kernel is implemented on top of the *gemm* routine of the Basic Linear Algebra Subprograms (BLAS). Having the same memory access pattern as the whole application, it reflects the whole computational workload. The problem size of the speed function of a process is represented by the area of the rectangle it handles. However, there are infinite number of rectangles of different shapes but of the same size, and the speed of process when handling these rectangles could be different. Since the partitioning algorithm used in this application arranges the submatrices to be as square as possible, for simplicity, speed functions are built with benchmarking only square matrices. In this chapter, we focus on data partitioning with respect to computational performance of processing unit. To this end, we do not model the communication between processing units. Instead, the shape and ordering of the submatrices assigned to processes are arranged so that the total communication volume is minimized [33].

3.4 Experimental Results

In this section, the proposed performance modeling and performance measurement methods are evaluated with parallel matrix multiplication on a heterogeneous multicore cluster. We experiment with a heterogeneous multicore platform, Grid5000 [20], which connects the clusters geographically distributed between 9 sites in France. In the experiment, 16 dedicated multicore nodes from four clusters, i.e. Paradent, Paramount, Parapide and Parapluie, of Rennes site, are used, specified in Table 3.2. The multicore nodes of different clusters are heterogeneous in terms of computing power and memory capacity. MPICH-1.2.7 is used for communication between processes.

To demonstrate the effectiveness of the data partitioning algorithm based on performance models built using the proposed method, it is compared with other data partitioning algorithms. In homogeneous data partitioning, the workload is evenly partitioned. In CPM-based data partitioning, the workload is partitioned in proportion to the constants that define the performance of processors. The constants are obtained in advance from speed measurements when some workload is distributed evenly between the processors.

In the experiment, the single-threaded *gemm* computational kernel is used. The

Table 3.2: Specifications of the four types of nodes used for experiments

Host	Paradent	Paramount	Parapide	Parapluie
Processor	Xeon L5420	Xeon 5418LV	Xeon X5570	Opteron 6164HE
Core Clock	2.50 GHz	2.33 GHz	2.93 GHz	1.70 Ghz
CPUs	2	2	2	2
Cores	8	4	8	24
L1 Cache	32 KB	32 KB	32 KB	64 KB
L2 Cache	6144 KB	4096 KB	256 KB	512 KB
L3 Cache			8 MB	10 MB
Memory Size	31 GB	8 GB	24 GB	48 GB
Memory BW	1.33 GT/s	1.33 GT/s	6.4 GT/s	6.4 GT/s

parallel matrix multiplication is executed with one process executing a computational kernel on each CPU core of the platform. In this case, each CPU core is modeled by an abstract processor. The speed of CPU cores of each node is measured when all cores in the node are executing the same amount of workload simultaneously. The speed is measured for wide range of problem sizes, then a speed function, $s_c(x)$, is built for each CPU core, where c is the number of CPU cores of the multicore node.

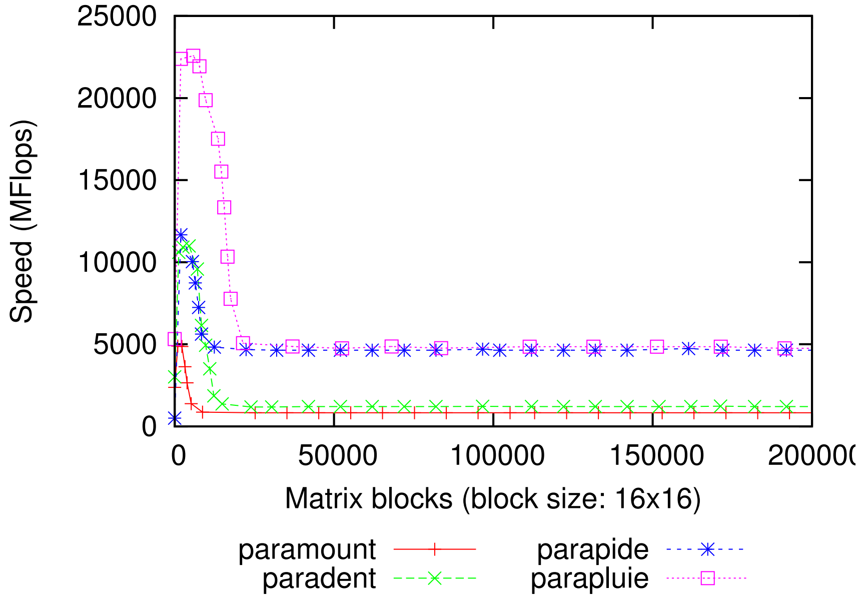


Figure 3.5: Functional performance models of the multicore nodes

Figure 3.5 shows the speed functions, $c \cdot s_c(x/c)$, of the four types of multicore nodes specified in Table 3.2, where c is the number of CPU cores of a multicore node. The x-axis represents the total number of matrix blocks, which represents the area of the updated matrix; and the y-axis represents the speed of the nodes. When the problem size is small enough to fit into the cache, there is relatively little memory traffic between caches and the main memory, and the highest performance is observed. When problem sizes cannot fit into the caches, the performance decreases with the increase of the problem size due to more load/restore operations and heavier memory traffic. For large problem sizes, memory traffic dominates the whole execution and therefore the application becomes memory-bound and the performance becomes stable.

With more CPU cores and larger cache, the Parapluie node has the best performance in the range of small problem sizes. But due to a lower CPU speed than the Paradent and Parapide node, the Parapluie node fails to gain a triple speedup. Memory bandwidth is the main bottleneck to achieve high performance when the problem size becomes very large. With advanced AMD *HyperTransport* and Intel *QuickPath Interconnect* technologies respectively, the Parapluie and Parapide nodes have a faster data transfer speed than the other two types of nodes. Therefore, they achieve a better performance in the range of large problem sizes. These four types of nodes represent a common heterogeneity of high performance computing platforms in the real world.

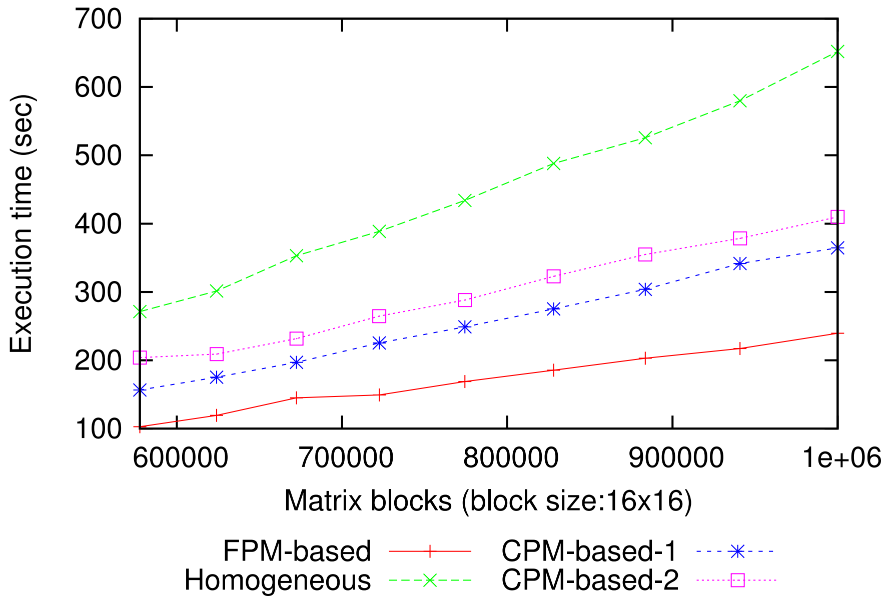


Figure 3.6: Execution time of parallel matrix multiplication on 16 multicore nodes with MFPM-based, CPM-based and homogeneous data partition schemes

Figure 3.6 shows the execution time of the parallel matrix multiplication application based on three different data partitioning schemes. We randomly select two sizes of matrix and run the computational kernel to generate the CPMs of these four types of multicore nodes. The CPM-based data partition 1 uses the CPMs when the size of matrix for benchmarking is 3000 blocks for each node, and the CPMs when the size is 7000 blocks is used in CPM-based data partition 2. Since FPM-based data partitioning scheme is more accurate and better capture different aspects of heterogeneity of heterogeneous multicore nodes than the other two schemes, it achieves the best performance.

Figure 3.6 demonstrates that the CPM-based data partition and Homogeneous data partition scheme are 2 times to 3 times slower than the FPM-based data partition scheme.

3.5 Summary

In this chapter, we present performance modeling and performance measurement methods on multicore systems, and extend the FPM-based data partitioning to heterogeneous clusters of multicore computers. We also present a procedure for performance measurement on heterogeneous clusters of multicore nodes. Based on proposed methods, functional performance models are built and are used as the input of data partitioning algorithms to balance the workload of a data parallel application on target platforms. Experimental results demonstrate that a data partitioning algorithm based on performance models built by the proposed methods are able to balance the workload of a data-parallel application on heterogeneous multicore platforms and achieve good performance.

Chapter 4

Data Partitioning on Heterogeneous Multicore and Multi-GPU Platforms

In this chapter, we focus on the problem of optimal data distribution of data parallel applications on heterogeneous multicore and multi-GPU platforms. Data parallel applications can be load balanced by applying data partitioning with respect to the performance of the platform's processing elements. However, load balancing on such platforms is complicated by several factors, such as contention for shared system resources, non-uniform memory access (NUMA), limited device memory and relatively low bandwidth of PCIe bus, which connects the host processor and the devices, etc. In this chapter, we propose methods of performance modeling and performance measurement on multicore and multi-GPU systems, and extend the FPM-based data partitioning algorithms to heterogeneous multicore and multi-GPU platforms.

4.1 Performance Modeling of multicore and multi-GPU Systems

On multicore and multi-GPU systems, where processing elements are coupled and share system resources, the speed of one processing element may depend on the load of others due to resource contention, therefore, they cannot be considered as independent processing elements and their speed cannot be measured separately.

In this work, we propose to model a multicore and multi-GPU system by a number of heterogeneous abstract processors. Each abstract processor represents a processing unit made of one or a group of processing elements executing one computational kernel of the application. Performance modeling methods of GPUs and CPU cores are presented in Section 4.1.1 and 4.1.2 respectively, as well as performance measurement methods that take into account the influence of resource contention on the performance of processing units. In these two sections, we focus on resource contention between only GPU processing units, and between only CPU processing units respectively, the impact of resource contention between CPU and GPU processing units on their performance is investigated in Section 4.1.3.

On target platforms, a parallel application can be executed in various configurations, depending on the computational kernels used in the application and their mapping to the processing elements of the platform. Nevertheless, in this work, we assume a parallel application will be configured following a set of simple rules, which we believe lead to efficient configurations: (i) all processing elements, including CPU cores and GPUs, will be utilized. (ii) if a multi-threaded CPU computational kernel is used, each group of CPU cores sharing memory will only execute a single such computational kernel. (iii) if a multi-GPU computational kernel is used, each group of GPUs sharing a PCIe link will only execute a single such computational kernel. In this work, we focus on the problem of optimal data distribution of data-parallel applications assuming that the configuration of the application is fixed. Comparison of different configurations and the problem of finding the optimal configuration of the application are out of the scope of this thesis.

For illustration, we use a multicore and multi-GPU server of NUMA architecture, Pluto, specified in Table 4.1. Pluto consists of eight NUMA nodes, with six cores and 16 GB memory each. It is equipped with a NVIDIA Tesla S2050 1U server, which consists of two pairs of GPUs. Each pair is connected by a PCIe switch that is linked to a separate NUMA node, that is, shares a PCIe link. The *gemm* kernels used for the CPU

Table 4.1: Specifications of the Pluto server *pluto.icl.utk.edu*

Hybird Server	Pluto (AMD CPU + NVIDIA GPU)	
Architecture	Opteron 6172	Tesla S2050
Core Clock	2.1 GHz	575 MHz
Number of Cores	8×6 cores	4×448 cores
Memory Size	8×16 GB	2667 MB
Mem. Bandwidth		4×148.4 GB/s
PCIe	$2 \times$ I/O hubs	$2 \times$ switches

and GPU are from libraries ACML 4.4 and CUBLAS 4.1 respectively.

4.1.1 Performance Modeling of GPUs

In GPU-accelerated multicore systems, a GPU is usually controlled by a host process executed on a dedicated CPU core. Since the GPU has a separate memory, input data are required to be transfered to the device for computation, and the result need to be copied back to the host memory afterwards. In this work, we model a GPU and its dedicated CPU core by an abstract processor, consequently, the data transfer time is included in the execution time during performance measurement. We measure the speed of the combined GPU processing unit (i.e. a GPU and its dedicated core) for a number of problem sizes to build its functional performance model. In general, the performance model can be defined only for the range of problem sizes that fit in the device memory. However, it can be extended for out-of-core applications, which can handle a larger amount of data stored in host memory through a large number of host-device data transfers.

As more and more GPUs added to GPU-accelerated multicore systems, additional PCIe lanes are required to maintain the available bandwidth to each GPU. There are two common strategies for increasing the available number of PCIe connections [116]. One approach is to introduce additional I/O hubs so that each GPU is connected to the host processor via a separate PCIe link. In this case, the performance models of GPU processing units can be built independently. Another approach is to utilize a PCIe switch. All data traffic traverses a single PCIe connection to the PCIe switch, and then is routed to GPUs connected to the PCIe switch. In this case, the performance models of GPU processing units cannot be built separately due to their contention for PCIe link. We propose to group GPU processing units by shared PCIe link. GPU processing units that share a PCIe link are grouped together and their performance is measured when all GPU processing units in the group are executing the same amount of workload simultaneously.

Wide use of multi-GPU systems encourages development of optimized computational kernels that could effectively distribute workload between multiple GPUs that share a PCIe link, minimize PCIe contention and overlap the host-device data transfers and device computations. If such a computational kernel is used, the dedicated CPU core and the multiple GPUs will make a combined processing unit, and will be modeled

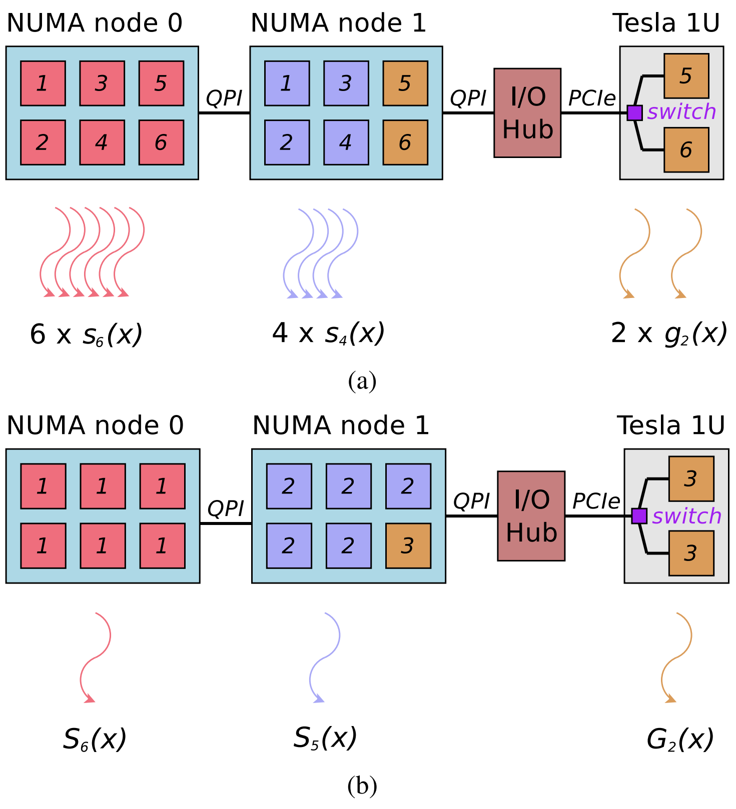


Figure 4.1: Performance modeling on a GPU-accelerated multicore server of NUMA architecture: (a) single-threaded and single-GPU computational kernels executed; each GPU handled by a dedicated CPU core (b) multi-threaded and multi-GPU computational kernels executed; two GPUs handled by a single dedicated CPU core

by an abstract processor.

Figure 4.1 shows a multicore and multi-GPU server of NUMA architecture executing a parallel application in two different configurations. In Figure 4.1(a), the single-GPU computational kernel is used, so each GPU and its dedicated CPU core make a combined processing unit, and are modeled by an abstract processor. One single-threaded computational kernel is executed on each of other CPU cores. As GPU processing units 5 and 6 on NUMA node 1 share a PCIe link, they cannot be considered independent. To measure the performance accurately, these two GPU processing units are grouped together. Their performance is measured when both of them are executing some workload simultaneously, thereby taking into account the contention for PCIe link. In Figure 4.1(b), the multi-GPU computational kernel is used, so the two GPUs and their dedicated CPU core make a combined processing unit, and are modeled by an abstract processor (processing unit 3 on NUMA node 1). One multi-threaded computational kernel is executed on each NUMA node with multiple threads. The performance of GPU processing units is measured for a wide range of problem sizes to build their functional performance models.

Three types of functional performance models for GPUs are defined as follows:

1. $g(x)$ approximates the speed of a combined processing unit made of a GPU and its dedicated CPU core that execute a single-GPU computational kernel, exclusively using a PCIe link. The speed $g(x) = x/t$, where x is the number of computation units, and t is the execution time.

2. $g_d(x)$ approximates the speed of one of d combined processing units, each made of a GPU and its dedicated CPU core. All combined processing units execute identical single-GPU computational kernels simultaneously and share a PCIe link. The speed $g_d(x) = x/t$, where x is the number of computation units executed by each GPU processing unit, and t is the execution time.

3. $G_d(x)$ approximates the speed of a combined processing unit made of d GPUs and their dedicated CPU core that collectively execute a multi-GPU computational kernel. The speed $G_d(x) = x/t$, where x is the total number of computation units processed by all d GPUs, and t is the execution time. $G_d(dx)/d$ is used to denote the average speed of a GPU.

Figure 4.2 shows the speed functions of a GPU processing unit built in different configurations on Pluto. Speed functions $g_1(x)$, $g_2(x)$, and $g_4(x)$ are built by executing a single-GPU *gemm* kernel per GPU processing unit on only one GPU processing unit, on two GPU processing units that share a PCIe link, and on two pairs of GPU processing units, each sharing a PCIe link. The dedicated CPU cores are located on NUMA nodes directly connected to the GPUs, therefore, the GPU processing units in a pair share not only PCIe but also memory. $G_2(2x)/2$ and $G_4(4x)/4$ approximate the average speed of a GPU, when one multi-GPU *gemm* kernel is executed on two GPUs that share a PCIe link, and on two pairs of GPUs, each sharing a PCIe link. In the last two configurations, only one CPU core is dedicated to GPUs. For all these experiments, the out-of-core *gemm* kernel implemented in CUDA is used. When the problem size exceeds the device

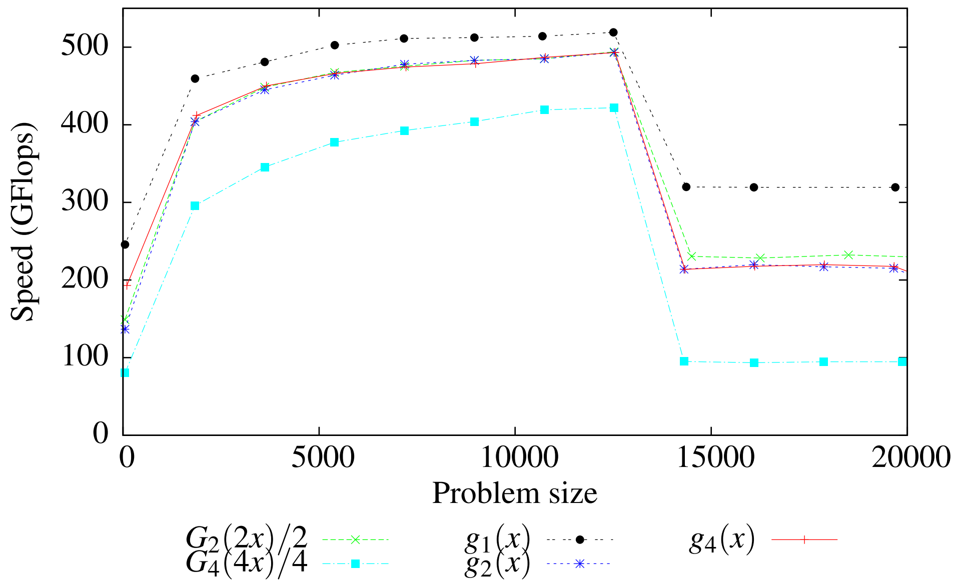


Figure 4.2: Speed functions of a GPU processing unit built in different configurations

memory, the performance of the kernel decreases due to more data transfers between the main and device memory. The multi-GPU version of the kernel is designed for GPUs that share a single data link and are handled by a single dedicated CPU core. The kernel schedules data transfers to eliminate PCIe contention between GPUs.

We can see that $g_2(x)$ is lower than $g_1(x)$, especially for large problem sizes. This indicates significant resource contention between two GPU processing units, dominated by PCIe but also including memory. There is no difference between $g_2(x)$ and $g_4(x)$, indicating there is no resource contention between the two pairs of GPU processing units. In the 4-GPU configuration, each pair of GPUs is connected to its own NUMA node, therefore, the performance of the pairs of GPU processing units can be measured independently. The performance degradation in $G_4(4x)/4$ compared to $G_2(2x)/2$ is caused by inappropriate use of the multi-GPU computational kernel, which is designed for GPUs sharing the same data link. In the 4-GPU configuration, due to the contention-free scheduling of data transfers, two data links are used alternately, remaining under-utilized during the execution of the kernel. In addition, the 4-GPU configuration uses the PCIe slots of two NUMA nodes but the memory of only one of them. As a result, one of the data links includes an extra QPI between NUMA nodes, which incurs extra communication overhead.

4.1.2 Performance Modeling of CPU cores

The method for performance modeling of CPU cores in a GPU-accelerated multicore system is the same as the performance modeling method for multicore systems in Chapter 3. The CPU cores are modeled by a number of abstract processors determined by the configuration of the parallel application. For example, if a single-threaded computational kernel is used, then on each CPU core one process executes a computational kernel, each CPU core will be modeled by an abstract processor. If a multi-threaded kernel is used, then one master process executes such a computational kernel on several CPU cores with multiple threads. Each group of CPU cores executing a computational kernel will make a combined processing unit, and will be modeled by an abstract processor. The master process is supposed to handles the placement and execution of threads, and workload distribution between threads.

To measure the performance accurately, CPU processing units are grouped by shared system resources so that the resources are shared within each group but not shared between groups. The performance of CPU processing units in a group is measured when all CPU processing units in the group are executing some workload simultaneously, thereby taking into account the influence of resource contention. The performance of CPU processing units are measured for a wide range of problem sizes to build their functional performance models.

Three types of functional performance models for CPU cores are defined as follows:

1. $s(x)$ approximates the speed of a uniprocessor executing a single-threaded computational kernel. The speed $s(x) = x/t$, where x is the number of computation units, and t is the execution time.

2. $s_c(x)$ approximates the speed of one of c CPU cores all executing the same single-threaded computational kernel. The speed $s_c(x) = x/t$, where x is the number of computation units executed by each CPU core, and t is the execution time.

3. $S_c(x)$ approximates the collective speed of c CPU cores executing a multi-threaded computational kernel. The speed $S_c(x) = x/t$, where x is the total number of computation units executed by all c CPU cores, and t is the execution time.

Figure 4.1 shows a multicore and multi-GPU server of NUMA architecture executing a parallel application in two different configurations. In Figure 4.1(a), one single-threaded computational kernel is executed on each CPU core of NUMA node 0. Therefore, each CPU core is modeled by an abstract processor representing a CPU processing unit made of one core. As the CPU cores of NUMA node 0 are tightly coupled and share memory, they cannot be considered independent. To measure the performance accurately, the CPU cores are grouped together. Their performance is measured simultaneously, thereby taking into account the influence of memory contention. As NUMA nodes are relatively independent in term of memory sharing, the performance of CPU cores of different NUMA nodes can be measured separately. In Figure 4.1(b), one multi-threaded computational kernel is executed on NUMA node 0 with multiple threads. Therefore, all CPU cores of NUMA node 0 make a combined CPU processing units and are modeled by an abstract processor.

Up to this point, we have focused on the memory contention between CPU processing units, and PCIe contention between GPU processing units separately. In the following section, we investigate the impact of resource contention between CPU and GPU processing units on their performance.

4.1.3 Impact of Resource Contention between CPU and GPU Processing Units

To achieve the maximum performance on a multicore and multi-GPU system, it is necessary to employ both CPUs and GPUs for computation. During the execution, while the CPU computational kernel performs computations using all levels of memory hierarchy, the GPU computational kernel mainly offload work to GPUs. Therefore, CPU and GPU processing units are heterogeneous in terms of computing power and as well as memory access pattern. As the CPU cores included in these two types of processing units share memory, they cannot be considered independent. For example, in Figure 4.1(a), CPU and GPU processing units share resources on NUMA node 1. The CPU and GPU processing units that share resources should be grouped together, and their performance should be measured simultaneously, thereby taking into account the influence of resource contention.

Figure 4.3 shows the speed functions of a combined CPU processing unit made of

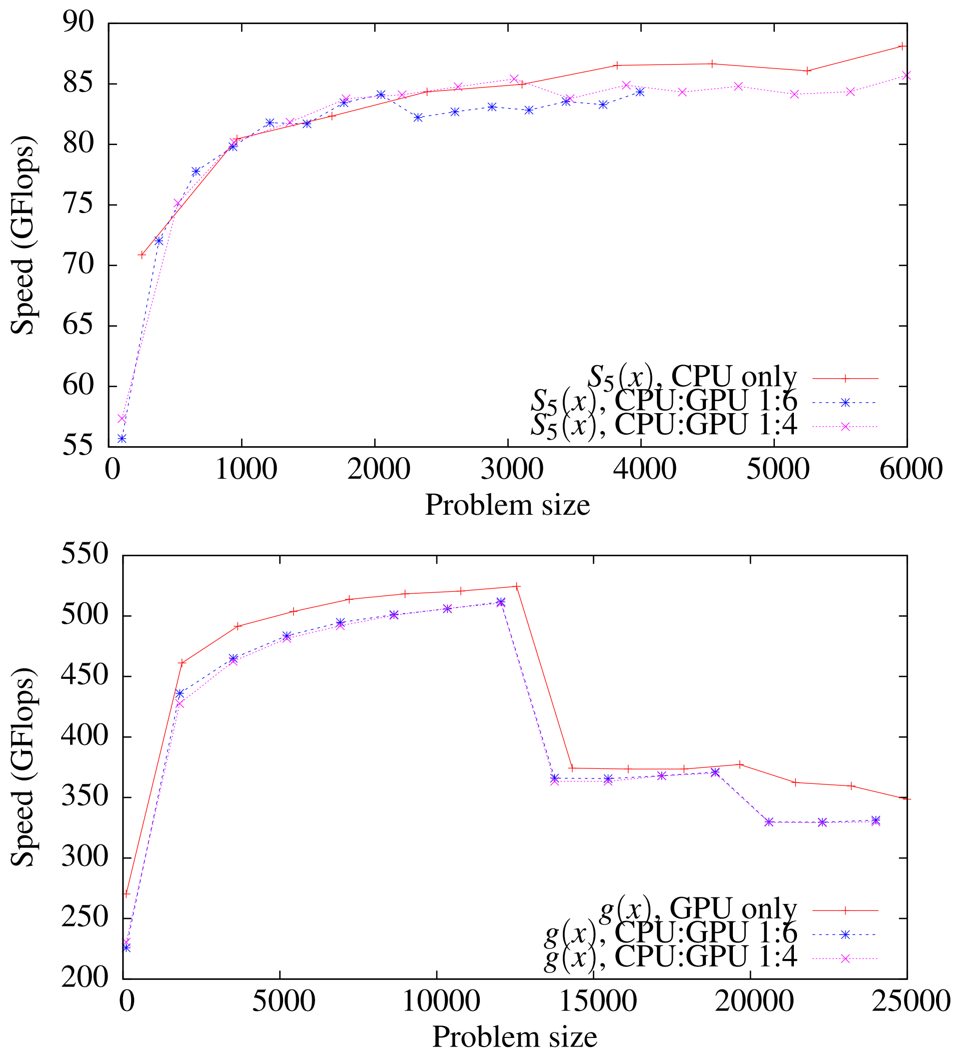


Figure 4.3: The impact of resource contention on the performance of the CPU (a) and GPU (b) processing units

five CPU cores of a NUMA node of Pluto, $S_5(x)$, built by executing the multi-threaded *gemm* kernel. The last CPU core of the NUMA node is dedicated to a GPU, therefore, they make a combined processing unit. The speed functions of the GPU processing unit, $g(x)$, built by executing the single-GPU *gemm* kernel in different configurations are also presented in the figure. The speed functions of the CPU and GPU processing units are built simultaneously with the influence of resource contention included. The speed functions built separately are also presented in the figure for comparison. The workloads are distributed in proportion to their speed measured independently. The distribution (1:6) corresponds to the relative speed when problem size fits in the device memory, while the distribution (1:4) to relative speed when problem size exceeds the device memory.

The resource contention has an obvious impact on the performance of the GPU processing unit, i.e. a performance reduction by 5% $\sim$ 10%, but exerts little impact on the combined CPU processing unit. In this case, the performance of processing units can be measured independently or simultaneously, which will not make too much difference. Therefore, functional performance models built independently are reasonably accurate for data partitioning. However, this is not always true. The amount of performance reduction due to resource contention may depend on the platform, application, and the implementation of the computational kernel. For example, in Chapter 5, the GPU performance of the computational kernel of a computational fluid dynamics application, which consists of a collection of linear algebra operations such as matrix-vector and vector-vector products, can be reduced by around 33% due to resource contention from other CPU cores of the same NUMA node. In that case, only performance models that are built with resource contention taken into account are accurate enough to be used for data partitioning.

4.1.4 Impact of NUMA Mapping and PCIe Contention

On a multicore system of NUMA architecture, data is transferred between NUMA nodes over links such as Intel QuickPath Interconnect (QPI), whose bandwidth is usually lower than memory bandwidth. Integration of multiple GPUs into multicore systems of NUMA architecture introduces complex performance phenomena. If the host process that handles the GPU is bound to a CPU core that resides in the NUMA node connected to the GPU directly through a I/O hub, the data processed by the GPU will only traverse links between the NUMA node, the I/O hub, and the GPU. Otherwise, the data processed by the GPU will traverse extra links between NUMA nodes, incurring extra communication overhead. In this work, these two types of configurations are referred to as *local* and *remote* respectively. In this section, we investigate the impact of process mapping in a GPU-accelerated multicore system of NUMA architecture on the performance of GPU processing units.

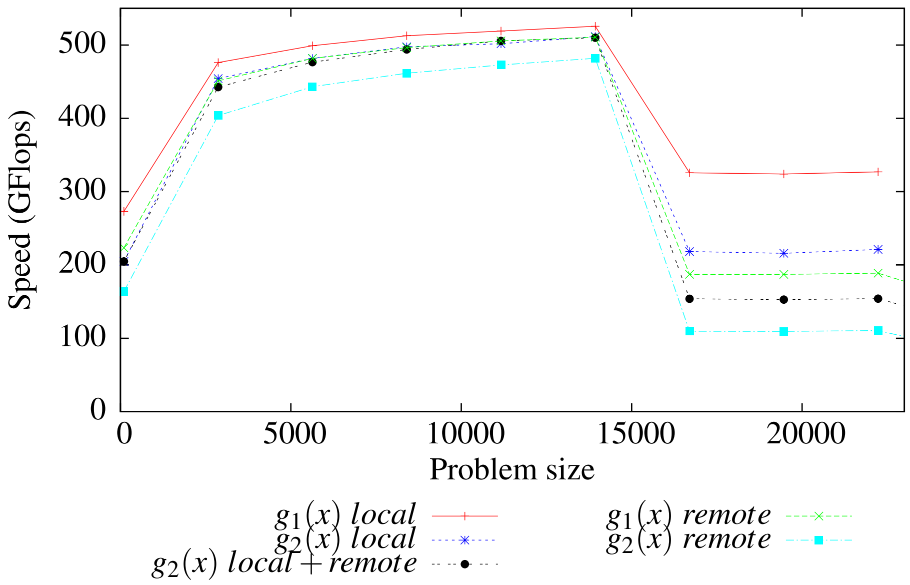


Figure 4.4: Speed functions of a GPU processing unit built in different configurations

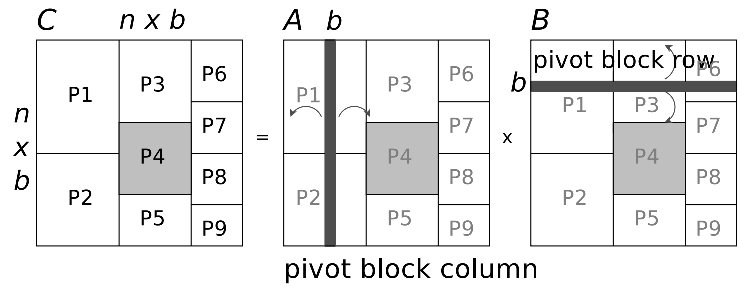
Figure 4.4 demonstrates the impact of NUMA mapping on the performance of a GPU processing unit, comprised of a CPU core and a GPU of Tesla S2050 deployed in Pluto. $g_1(x)$ is built by executing one single-GPU *gemm* kernel, which uses exclusively the PCIe link and the memory of a local or remote NUMA node. $g_2(x)$ is built by executing two single-GPU kernels simultaneously on two GPU units that share the PCIe link and the memory of the same NUMA node, local or remote. In the remote configuration, the GPU units also share an extra QPI link to the remote NUMA node. Speed function $g_2(x)$ is also built in the configuration, when the dedicated CPU cores are located on different NUMA nodes, which is denoted as *local* + *remote*. In this case, the processing units share PCIe but do not share memory.

The difference between speed functions $g_1(x)$ and $g_2(x)$, built in local or remote configuration, reflects the performance degradation due to the contention for PCIe link and possibly also for main memory of the same NUMA node. Significant difference is observed for large problem sizes when a large number of data transfers are required. Communication overhead between NUMA nodes can be estimated by the difference between $g_1(x)$ in *local* and *remote* configurations. The combined effect of both phenomena is reflected by the $g_2(x)$ functions in different configurations.

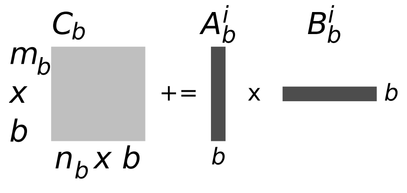
4.2 Heterogeneous Parallel Matrix Multiplication

In this section, firstly, we refresh the data parallel application, i.e. heterogeneous parallel matrix multiplication [33], which will be used in experiments; then, we discuss the implementation of the computational kernel of this application for GPUs.

Heterogeneous parallel column-based matrix multiplication has been described in Section 3.3 in detail. The heterogeneous parallel application was originally designed for a heterogeneous network of uniprocessors. It takes the functional performance models of heterogeneous processors as input, partitions the matrices using the FPM-based data partitioning algorithm, and then performs the blocked matrix multiplication using vendor-optimized *gemm* kernels. In this algorithm each process is responsible for calculations associated with a rectangular submatrix. These rectangles are arranged into columns and the area of the rectangles is proportional to the speed of the processing element upon which the process is running, as shown in Figure 4.5. A communication minimizing algorithm proposed in [11] uses this column-based partitioning and finds the shape and ordering of these rectangles such that the total volume of communication for parallel matrix multiplication is minimized. The computational kernel of this appli-



(a)



(b)

Figure 4.5: Heterogeneous parallel column-based matrix multiplication. (a) One step of the algorithm. (b) Computational kernel.

cation is matrix multiplication, as shown in Figure 4.5(b). On CPU-based computing systems, the *gemm* kernel from optimized libraries like BLAS can simply be directly invoked to implement the computational kernel. However, the computational kernel for GPUs should be carefully designed, taking into account the separate memory address spaces of CPU and GPU, and the limited memory capacity of the GPU.

The following two sections presents the design of computational kernels for a single GPU with its own PCIe link, and for multiple GPUs that share a PCIe link respectively. To overcome the limited memory capacity of GPUs, out-of-core computational kernels are designed. A static scheduling strategy, which is designed to avoid PCIe contention and improve the utilization of the shared PCIe bus of multiple GPUs, is discussed.

4.2.1 Design of the Single-GPU Computational Kernel

In the single-GPU computational kernel, when matrices A_b, B_b, and C_b fit in the device memory, we allocate a set of three data buffers in the device memory for storing them respectively. At the first iteration of the application, all three submatrices are transferred to the device and stored in the buffers. The *gemm* kernel from a vendor-optimized library, e.g. CUBLAS, is then invoked to update C_b. In this application, the computational kernel will be executed multiple times iteratively with different pivot rows and pivot columns for updating C_b. At each iteration, A_b and B_b will be transferred to the device memory, and C_b will be updated and accumulated in the device memory. To reduce the communication overhead over the PCIe bus, the updated C_b will only be transferred back to the host memory at the last iteration of the application.

Due to the memory limit of the GPU, the GPU *gemm* kernel from vendor-optimized libraries cannot be used straightforwardly in cases when the matrix size is too large. To exploit the computing power of GPUs, we developed an out-of-core computational kernel for the purpose of demonstration. In the out-of-core implementation, C_b is split in two dimensions into a number of rectangular blocks to be updated. Accordingly, A_b and B_b are split in one dimension into horizontal and vertical slices respectively. The data required for updating a matrix block of C_b is small enough to fit in the device memory. Matrix blocks of C_b will be updated one by one on the device by invoking the *gemm* kernel from a vendor-optimized library. During each iteration, multiple transfers of the matrix blocks between the host memory and device memory are required, leading to a significant communication cost. To reduce the communication overhead, we overlap data transfers and *gemm* executions. To this end, two sets of data buffers are allocated in the device memory, each for updating one matrix block of C_b. While the update of one matrix block of C_b is being conducted using the data from one set of buffers, data transfers can be performed between the host memory and the other set of data buffers.

Algorithm 1 illustrates the implementation of the out-of-core computational kernel in CUDA [96]. The *gemm* kernel used is from the vendor-optimized library CUBLAS [95], an implementation of BLAS on top of the CUDA runtime. In this implementation, submatrix C_b is partitioned into h rows and w columns, thus producing $h \times w$ rectangular blocks c_{ij}, A_b is horizontally split into h slices a_i, and B_b is vertically split into w slices b_j, $0 \leq i < h$, $0 \leq j < w$. A matrix block c_{ij} is of the same height as matrix block a_i, and of the same width as matrix block b_j. The matrix blocks of C_b are updated column by column. In each column, the matrix blocks are updated one by one from the top to the bottom. The matrix blocks are updated using multiple CUDA ***streams*** so that in different streams data transfers and *gemm* executions can be executed concurrently, and data can be transferred concurrently in both directions (if hardware supports). To leverage concurrent data transfers and overlapping of data transfers and *gemm* executions, five buffers, $\bar{A}_0$, $\bar{A}_1$, $\bar{B}_0$, $\bar{C}_0$, $\bar{C}_1$, are allocated for storing matrix blocks in the device memory using its maximum capacity. More specifically, $\bar{A}_0$ and $\bar{A}_1$ are allocated to store one matrix block of A_b each. $\bar{B}_0$ is allocated to store one matrix block of B_b. $\bar{C}_0$ and $\bar{C}_1$ are allocated to store one matrix block of C_b each. When one matrix block c_{ij} is being updated using data stored in a set of buffers, e.g. $\bar{A}_0$, $\bar{B}_0$, $\bar{C}_0$, in one stream, data transfers can be performed between the host memory and the other set of data buffers, e.g. $\bar{A}_1$, $\bar{C}_1$ in another stream simultaneously. Note that data in the buffer $\bar{B}_0$ will be reused for updating matrix blocks of every column of C_b, and will be renewed once for each column. To make sure data stored in device buffers will not be renewed until *gemm* executions that operate on the data have completed, we create three CUDA ***event*** arrays. Note that synchronization to an event recorded in one stream from another given stream will make all future operations submitted to the given stream wait until the event reports completion before beginning execution.

At the beginning of Algorithm 1, the transfer operations of a_0, b_0, and c_{00} to device buffers $\bar{A}_0$, $\bar{B}_0$, and $\bar{C}_0$ are submitted to stream 0. Then, three steps repeat until the last matrix block of C_b is updated:

Algorithm 1: Implementation of the out-of-core kernel for a single GPU

```
for $j = 0$ to $w - 1$ do
    for $i = 0$ to $h - 1$ do
        $k = i + j \cdot h$;    $l = k\%2$;
        $i' = (k+1)\%h$;    $j' = (k+1)/h$;    $l' = (k+1)\%2$;
        if $k == 0$ then transfer operations of matrix blocks $a_i$, $b_j$ and $c_{ij}$ to device
        buffers $\bar{A}_l$, $\bar{B}_0$ and $\bar{C}_l$ submitted to $stream[k]$

        if $k < h \cdot w - 1$ then
            // execute cublas gemm to update matrix block $c_{ij}$
            gemm execution for updating $c_{ij}$ submitted to $stream[k]$
            cuda event $event\_a[k]$ recorded in $stream[k]$
            if $i == (h-1)$ then cuda event $event\_b[k]$ recorded in $stream[k]$

            // transfer blocks of $a_{i'}$, $b_{j+1}$ and $c_{i'j'}$ to device buffers
            if $k > 0$ then sync with $event\_a[k-1]$ recorded in $stream[k+1]$

            transfer operation of $a_{i'}$ to buffer $\bar{A}_{l'}$ submitted to $stream[k+1]$
            if $i == (h-1)$ then
                if $j == w-1$ then sync with $event\_b[k]$ recorded in $stream[k+1]$

                transfer operation of $b_{j+1}$ to buffer $\bar{B}_0$ submitted to $stream[k+1]$
            end
            if $k > 0$ then sync with $event\_c[k-1]$ recorded in $stream[k+1]$

            transfer operation of $c_{i'j'}$ to buffer $\bar{C}_{l'}$ submitted to $stream[k+1]$
            // transfer updated block $c_{ij}$ stored in buffer $\bar{C}_{\bar{l}}$ back to the host
            transfer operation of updated $c_{ij}$ back to host submitted to $stream[k]$
            cuda event $event\_c[k]$ recorded in $stream[k]$
        else
            // update the last block $c_{ij}$, $i = h-1$, $j = w-1$ on device
            gemm execution for updating $c_{ij}$ submitted to $stream[k]$
            transfer operation of updated $c_{ij}$ back to host submitted to $stream[k]$
        end
    end
end
```

1. The update of matrix block c_{ij}, i.e *gemm* execution, is submitted to *stream* k, $k = i + jh$. Note that the input data, i.e. a_i, b_j, c_{ij}, has already been transferred to a set of buffers $\bar{A}_l$, $\bar{B}_0$, $\bar{C}_l$, $l = k\%2$. Then one event in *event_a* is recorded asynchronously to ensure that data stored in buffer $\bar{A}_l$ will not be renewed until the matrix block c_{ij} is updated. If c_{ij} is the matrix block at the bottom of column j of C_b, one event in *event_b* is recorded asynchronously to ensure that data in this buffer will not be renewed until the matrix block c_{ij} is updated.

2. The transfer operations of the input data for the update of the next matrix block $c_{i'j'}$, i.e. matrix blocks $a_{i'}$, $c_{i'j'}$, where i' is its row index and j' is its column index, from the host memory to buffers $\bar{A}_{l'}$, $\bar{C}_{l'}$, $l' = (k+1)\%2$, are submitted to *stream* $k+1$. Note that data in $\bar{B}_0$ is reused for updating matrix blocks of one column of C_b. If $c_{i'j'}$ is at the top of the next column, the transfer operation of matrix block $b_{j'}$ to buffer $\bar{B}_0$ is also submitted to *stream* $k+1$. An asynchronous synchronization function is called before each data transfer operation. Note that data transfers will not be actually executed until the corresponding recorded events have completed.

3. The transfer operation of the updated matrix block c_{ij} stored in device buffer $\bar{C}_l$ back to the host memory is submitted to stream k. Note that operations submitted to the same stream will be executed in issue-order on the device. Then one event in *event_c* is recorded to ensure data stored in buffer $\bar{C}_l$ will not be renewed until the updated block c_{ij} has been transferred back to the host memory.

At the end, the last block $c_{(h-1)(w-1)}$ is updated on the device and then transferred back to the host memory. All operations are asynchronous with respect to the host, which are submitted to streams first and then executed concurrently.

To reduce communication overhead incurred by transferring matrix blocks between the host and device memory, the last two matrix blocks of C_b will not be transferred back to the host memory after they are updated. Instead, the updating order of the matrix blocks of C_b will be reversed in the next iteration of the application. In this way, the first two matrix blocks of C_b to be updated in the next iteration, i.e. the last two matrix blocks in this iteration, will be already stored in the device buffers, saving data transfers of two matrix blocks at the end of this iteration and data transfers of two matrix blocks at the beginning of the next iteration. For the sake of brevity, this optimization

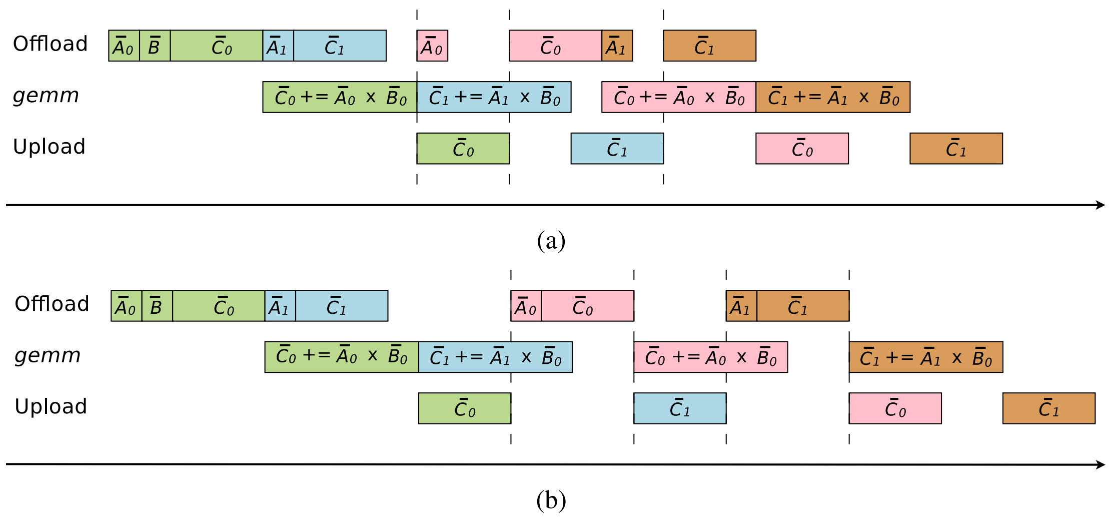

Offload
gemm
Upload
$\bar{A}_0$ $\bar{B}$ $\bar{C}_0$ $\bar{A}_1$ $\bar{C}_1$
$\bar{C}_0 += \bar{A}_0 \times \bar{B}_0$
$\bar{C}_1 += \bar{A}_1 \times \bar{B}_0$
(a)
(b)

Figure 4.6: Timeline of the execution of algorithm 1 with matrix C_b split into four matrix blocks (a) on a device with two copy engines which supports concurrent data transfers and overlapping of data transfers and kernel executions (b) on a device with only one copy engine which only support overlapping of data transfers and kernel executions in one direction. The operations marked by the same color are submitted to the same stream, otherwise are submitted to different streams.

is not presented in the pseudocode. In addition, the dimensions of the matrix blocks are ensured to be multiples of 32, taking into account the impact of memory alignment issues of CUDA on the Level 3 BLAS implementation of CUBLAS [9].

This implementation of the single-GPU computational kernel can be used on a GPU with one copy engine which only support overlapping of kernel execution and data transfer in one direction, and on a more advanced GPU with two copy engines which support both concurrent data transfers in two directions and overlapping of kernel executions and data transfers. Note that in this implementation we only execute one kernel on the device at a time. Although this computational kernel can be directly used on all generations of GPUs, it can be improved to support concurrent kernel executions supported by modern advanced GPUs for better performance in the future work.

Figure 4.6 illustrates the overlapping behavior of algorithm 1. In this example, C_b is split into 2×2 matrix blocks, and A_b and B_b are split into 2 matrix blocks each accordingly. Four streams are created for updating the four matrix blocks of C_b concurrently. As shown in the figure, each row consists of a number of operations of the same type. The first row shows data transfer operations from the host memory to device buffers $\bar{A}_0$,

$\bar{A}_1$, $\bar{B}_0$, $\bar{C}_0$, $\bar{C}_1$ in four streams. In the figure, CUDA operations submitted to the same stream are marked by the same color, otherwise are marked by different colors. The second row shows *gemm* executions operating on data stored in device buffers, and the third row shows data transfer operations of updated matrix blocks from device buffers $\bar{C}_0$, $\bar{C}_1$ to the host memory. While CUDA operations submitted to the same stream will be executed in issue-order, operations submitted to different streams can be executed concurrently. The amount of communication that could be hidden depends on the ratio of the data transfer time to the *gemm* execution time. If the *gemm* execution dominates the total execution time, then, the smaller the ratio, the more communication could be overlapped. However, if the communication over the PCIe bus dominates the total execution time, the communication would be always the bottleneck.

In the parallel matrix multiplication using the out-of-core computational kernel, the total volume of communications between the host and device memory is determined by the blocking factor b. In each iteration of parallel matrix multiplication, the submatrix C_b to be updated by a device will be transferred between the host and device once in each direction, by transferring the matrix blocks serially. The total number of transfers of C_b doubles the number of iterations of the application. By increasing the blocking factor, the number of iterations can be decreased, resulting in less total volume of communications. The performance will be improved since the data transfer time occupies a large part of the GPU execution time. Meanwhile, with a larger b, all processing elements perform better, benefiting from the optimized *gemm* kernels, and the communication operations (such as broadcast) between processing elements decrease. However, too large a blocking factor will result in a coarse-grained partitioning of matrices, which may reduce the level of parallelism and leave less opportunity to balance the workload. Thus, the blocking factor b should be tuned depending on platforms and parallel routines to achieve a better performance, which is out of the scope of this thesis.

Figure 4.7 presents the speed functions built with different modifications of the computational kernel on NVIDIA GeForce GTX680 deployed in the Ig server, specified in Table 4.2. The computational kernels are executed in single precision with $b = 640$ on a dedicated core, while other cores stay idle. Version 1 is designed for problem sizes that fit in the device memory. At each iteration, A_b, B_b, and C_b are transferred to GPU in full; the updated C_b is transferred back to the host memory and not reused. In version 2 and 3, C_b is accumulated in the device memory when problem sizes fit in the device

Table 4.2: Specifications of the Ig server *ig.icl.utk.edu*

Hybird Server	Ig (AMD CPU + NVIDIA GPU)		
Architecture	Opteron 8439SE	GF GTX680	Tesla C870
Core Clock	2.8 GHz	1006 MHz	600 MHz
Number of Cores	4×6 cores	1536 cores	128 cores
Memory Size	4×16 GB	2048 MB	1536 MB
Mem. Bandwidth		192.3 GB/s	76.8 GB/s
Number of PCIe		1	1

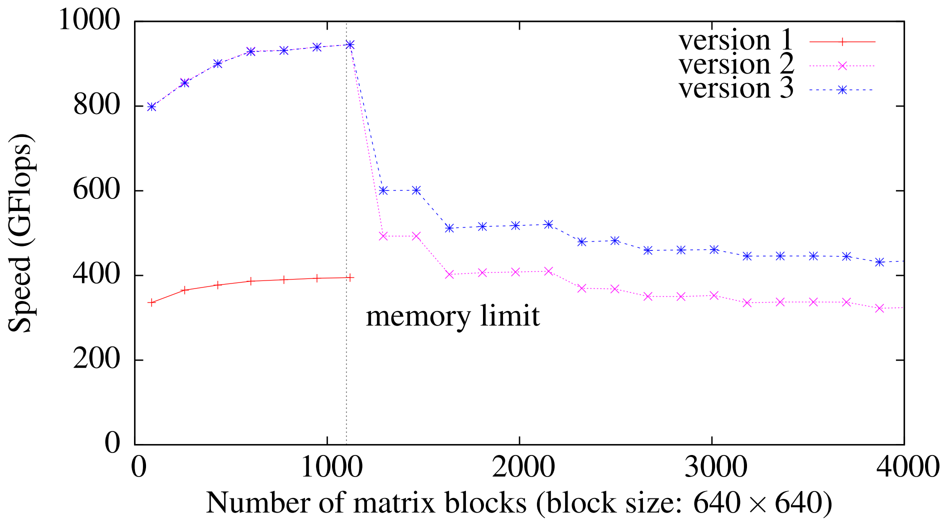


Figure 4.7: Speed functions of GeForce GTX680 on Hybrid system Ig

memory. Hence, the data transfers of C_b are excluded from the speed measurement. As a result, the performance doubles that of version 1 in the range of small problem sizes. When problem sizes exceed the device memory, the out-of-core computations are enabled. This implementation requires many data transfers of matrix blocks of C_b to and from the device memory, which explains the performance drop in this range of problem size. In version 2, concurrent data transfers and overlapping of data transfers with computations in the device are disabled, while in version 3 these two optimization techniques are enabled. Therefore, the performance of GeForce GTX680 executing the kernel of version 3 improves by up to 30% in the range of large problem sizes.

4.2.2 Design of the Multi-GPU Computational Kernel

In this section, we briefly describe the design of the multi-GPU computational kernel for a GPU server consisting of multiple identical GPUs that share one PCIe link.

When problem sizes fit in the sum of the device memory of GPUs, submatrix C_b is partitioned into a number of rectangular matrix blocks, each to be updated by one GPU. Accordingly, A_b and B_b are partitioned into slices. We allocate a set of three data buffers for storing one matrix block of C_b, and one matrix block of A_b, i.e. the pivot column, and one matrix block of B_b, i.e. the pivot row, respectively. Matrix blocks are transferred to devices and stored in these data buffers. The *gemm* kernel from a vendor-optimized library, e.g. CUBLAS, is then invoked on each device to update one matrix block of C_b. In the parallel matrix multiplication, the computational kernel will be executed multiple times iteratively with different pivot columns and pivot rows for updating C_b. At each iteration, matrix blocks of A_b and B_b will be transferred to the devices, and the matrix blocks of C_b will be updated and accumulated in the device memory. To reduce the communication overhead over the PCIe bus, the updated matrix blocks of C_b will only be transferred back to the host memory at the last iteration of the application.

When problem sizes exceed the sum of the device memory of GPUs, submatrix C_b is partitioned into a number of rectangular matrix blocks, which are then distributed to GPUs to be updated. Submatrix C_b is split into $h \times w$ matrix blocks c_{ij}, A_b is horizontally split into h matrix blocks a_i, and B_b is vertically split into w matrix blocks b_j accordingly, $0 \leq i < h, 0 \leq j < w$. The data required for updating a matrix block of C_b is small enough to fit in the device memory of one GPU. Let d denote the number of GPUs sharing the PCIe bus, and r denote the id of a GPU, $0 \leq r < d$. Matrix blocks of column j of C_b are sent to GPU r for update, $r = j \mod d$. Each GPU updates one or multiple columns of matrix blocks. All GPUs will update matrix blocks of C_b simultaneously. However, matrix blocks assigned to an individual GPU are updated one by one from the top to the bottom in each column, column by column.

The matrix blocks of C_b are updated using multiple CUDA ***streams*** so that in different streams data transfers and *gemm* executions can be executed concurrently, and data can be transferred concurrently in both directions (if hardware supports). Similar to Algorithm 1, to leverage concurrent data transfers and overlapping of data transfers and *gemm* executions, five buffers, $\bar{A}_0, \bar{A}_1, \bar{B}_0, \bar{C}_0, \bar{C}_1$ are allocated in the device memory of each GPU using its maximum capacity for storing matrix blocks. Also, three ***event*** arrays, *event_a*, *event_b* and *event_c*, are created to ensure data stored in device buffers will not be renewed until all operations operating on the data stored in these buffers have completed. In addition, two global event arrays, *event_send* and *event_recv*, are created to ensure that data transfers issued on device $r+1$ in a stream will not be executed until the data transfers issued on device r in the same stream in the same direction have completed, $0 \leq r < d-1$, except that device 0 will wait until device $d-1$ completes data transfer operations. Data transfers from the host to the device memory of all GPUs are scheduled so that the shared PCIe link is used exclusively to communicate by one device at a time, thereby, avoiding PCIe contention. For updating one matrix block of C_b, the transfer of the input data of *gemm* execution and the transfer of output data are submitted in the same stream and will be executed in issue-order. As the transfer operations of the input data are scheduled so that one operation will be executed at a time, the transfer operations of the output data will be executed one by one consequently.

Figure 4.8 illustrates the overlapping behavior of the multi-GPU computational kernel. In this example, the multi-GPU server consists of two GPUs sharing one PCIe bus, and C_b is split into 4×2 rectangular matrix blocks. A_b and B_b split into smaller matrix

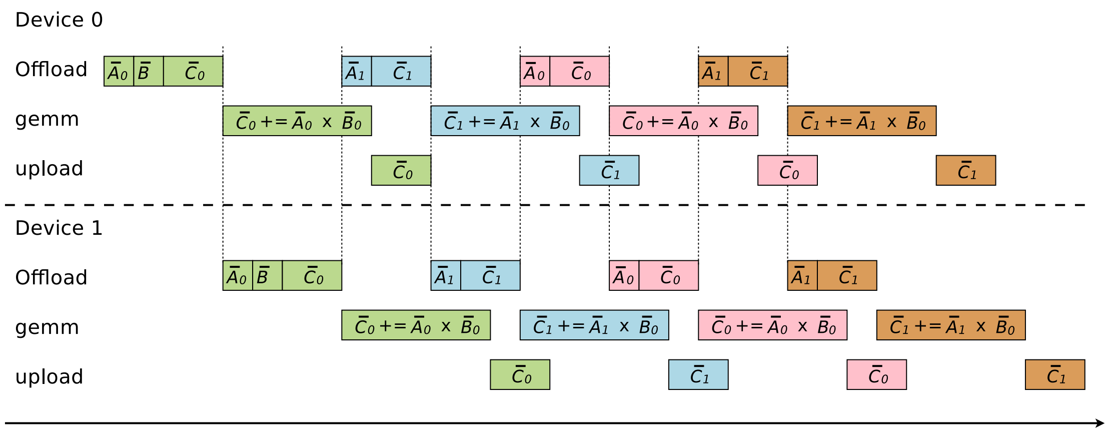


Figure 4.8: Timeline of the execution of the out-of-core multi-GPU computational kernel. C_b is partitioned into 4×2 rectangular matrix blocks, which are then distributed to two devices for update. For each device, CUDA operations marked by the same color are submitted to the same stream, otherwise are submitted to different streams. Data transfer operations are scheduled so that only one operation will be executed at a time, in order to avoid PCIe contention.

blocks accordingly. Four streams are created on each device for updating blocks of C_b concurrently. The upper three rows of the timeline demonstrate the overlapping behavior of device 0 while the lower three rows shows that on the other device. Each row consists of a number of operations of the same type. The first row shows data transfer operations from the host memory to device buffers $\bar{A}_0$, $\bar{A}_1$, $\bar{B}_0$, $\bar{C}_0$, $\bar{C}_1$ in four streams. In the figure, for each GPU, CUDA operations submitted to the same stream are marked by the same color, otherwise are marked by different colors. The second row shows *gemm* executions operating on data stored in device buffers, and the third row shows data transfer operations of updated matrix blocks from device buffers $\bar{C}_0$, $\bar{C}_1$ to the host memory. While CUDA operations submitted to the same stream will be executed in issue-order, operations submitted to different streams can be executed concurrently.

In this example, the data transfer time and kernel execution time are comparable. As we can see, the PCIe bus is always busy with data transfers in the direction from the host to the device, until the last block is sent to the device buffer. The total execution time can be estimated by the sum of the time for transferring all matrix blocks from the host to the two GPUs, the *gemm* execution time for updating the last block on GPU 1, and the time for transferring it back to the host. This multi-GPU computational kernel is efficient in terms of the utilization of the PCIe bus, on the premise that the throughput speed of the PCIe does not vary wildly. In cases when the kernel execution time dominates the total execution time, this multi-GPU computational kernel is able to achieve good computation parallelism since kernel executions are performed independently on both GPUs. In cases when data transfer time dominates, the communication over PCIe link will be the bottleneck. The speed functions built by executing the multi-GPU computational kernel in different configurations are shown in Figure 4.2.

4.3 Experimental Results

In this section, the proposed performance modeling and performance measurement methods are evaluated with a fundamental data parallel application, namely, parallel matrix multiplication. Firstly, experimental results of data partitioning based on pre-built FPMs on a multicore and multi-GPU server are presented; then, we experiment with a heterogeneous GPU-accelerated multicore cluster. Instead of pre-built FPMs, partial FPMs that are built dynamically at application run-time to a sufficient level of accuracy to achieve load balancing are used. Experimental results of data partitioning based on different performance models are presented.

4.3.1 FPM-based Data Partitioning on Hybrid Multicore and Multi-GPU Servers

The platform used for experiment is a multicore and multi-GPU server, Ig, specified in Table 4.2. This server consists of four identical NUMA nodes, with 6 cores and 16 GB local memory each, and is accelerated by two different NVIDIA GPUs.

Table 4.3: Execution time of the application on hybrid system Ig

Problem size	CPUs only	GTX680 only	Hybrid-FPM
40×40	99.5	74.2	26.6
50×50	195.4	162.7	77.8
60×60	300.1	316.8	114.4
70×70	491.6	554.8	226.1

Table 4.3 shows the execution time of the heterogeneous matrix multiplication application in different configurations on Ig. In the experiment, single-threaded and single-GPU computational kernels are used. Each GPU and its dedicated CPU core make a combined processing unit. A speed function, $g(x)$, is built for each GPU processing unit. The two CPU cores dedicated to GPUs are from two different NUMA nodes. A speed function, $s_5(x)$, is built for CPU cores in these two NUMA nodes. A speed function, $s_6(x)$, is built for CPU cores in other two NUMA nodes. Column 1 shows the problem sizes of the application, i.e. the numbers of matrix blocks of size 640×640. Column 2 shows the execution time of the application executed using all 24 CPU cores, with workload distributed to CPU cores homogeneously. Column 3 shows the execution time of the application executed on GeForce GTX680. Column 4 shows the execution time of the application executed using all CPU cores and GPUs, with workload partitioned to CPU cores and GPUs by FPM-based data partitioning algorithm. Experiment results shows that the GeForce GTX680 outperforms 24 CPU cores when problem sizes fit in the device memory. When problem sizes exceed the device memory, CPUs perform better. These variations are captured by functional performance models, therefore, the FPM-based data partitioning algorithm is able to balance computations under all

Table 4.4: Heterogeneous data partitioning on hybrid system Ig

Problem size	CPM-based				FPM-based			
$n \times n$	$G1$	$G2$	$S5$	$S6$	$G1$	$G2$	$S5$	$S6$
40×40	928	226	105	120	1000	210	95	102
50×50	1460	352	160	186	1250	429	190	222
60×60	2085	501	235	270	1627	657	295	342
70×70	2848	677	320	366	2250	806	425	504

problem sizes and results in good performance.

To demonstrate the accuracy of the FPM-based data partitioning, we compare it with data partitioning based on the constant performance model (CPM). In CPM-based data partitioning, the speed of each processor is represented by a constant, which is obtained in advance from the speed measurements. In Table 4.4, we present the results of the CPM- and FPM-based partitioning algorithms with different problem sizes. Column 1 shows the problem sizes of the application. Columns $G1$ and $G2$ shows the numbers of matrix blocks distributed to GeForce GTX680 and Tesla C870 respectively. Columns $S5$ shows the total number of matrix blocks partitioned to the other five cores in the NUMA node in which one CPU core is dedicated to a GPU. Columns $S6$ shows the total number of matrix blocks distributed to all CPU cores in the NUMA node in which no CPU core is dedicated to GPUs.

According to the speed functions, GeForce GTX680 is around 9 times faster than a NUMA node when the problem size fits in the device memory (40×40), and around $6 \sim 4$ times faster when problem sizes exceed the device memory (from 50×50 to 70×70). As we can see in Table 4.4, the CPM-based data partitioning resulted in overloading GeForce GTX680, starting from problem size 50×50. For example, the ratio of the number of matrix blocks partitioned to GeForce GTX680 and a NUMA node is nearly 8 when the problem size is 70×70, which is far from the balanced ratio of $6 \sim 4$. The reason for overloading is that the CPM-based algorithm used inaccurate performance models. The speed of GPU used for data partitioning is measured when the problem size happened to fit in the device memory, therefore it can not represent the speed when problem sizes exceed the device memory. As the functional performance model captures the change of performance in a wide range of problem sizes accurately, the FPM-based partitioning algorithm is able to balance the load and deliver good performance.

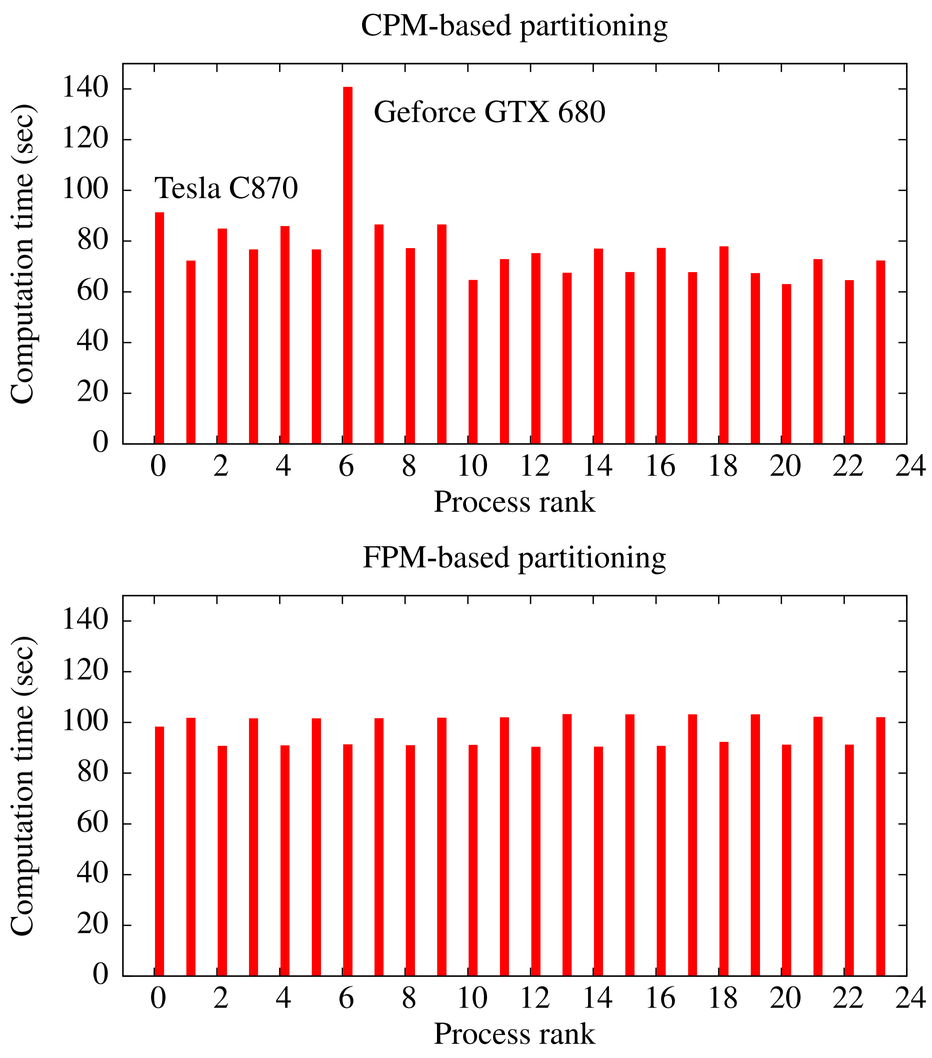


Figure 4.9: The computation time of each process on Ig

Figure 4.9 illustrates the computation time (communication time between processes excluded) of each process when problem size is 60×60, and the workload is partitioned by CPM- and FPM-based partitioning algorithms respectively. In both experiments, process 0 and 6 are bound to CPU cores dedicated to Tesla C870 and GeForce GTX680 respectively. With the CPM-based data partitioning, GeForce GTX680 took a longer time than other processes to finish its job because it is overloaded. The CPM-based data partitioning fails to balance the workload. The FPM-based data partitioning algorithm achieves good load balance and reduces the total computation time by around 40%.

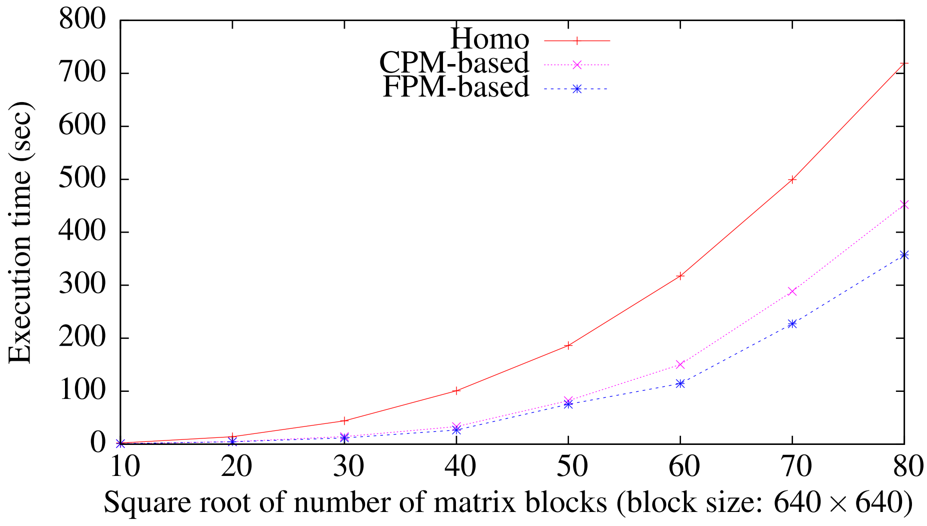


Figure 4.10: Execution time of the parallel matrix multiplication application on hybrid system Ig with different data partitioning algorithms

Figure 4.10 shows the execution time (including communication time between processes) of the heterogeneous parallel matrix multiplication application when the workload is distributed by different data partitioning algorithms. The execution of the application based on homogeneous partitioning (data distributed evenly) is unbalanced, being dominated by the slowest processing elements (CPU cores). Both the CPM-based and FPM-based data partitioning are able to balance the workload when problem sizes are relatively small. However, starting from problem size 50×50, the CPM-based algorithm fails to balance the workload and the application takes longer time to complete than the application based on the FPM-based algorithm. The FPM-based data partitioning algorithm reduces the execution time of the application over the CPM-based and homogeneous partitioning algorithms by up to 21% and 50% respectively, in the range of large problem sizes.

4.3.2 Partial FPM-based Data Partitioning on Heterogeneous Multicore and Multi-GPU Clusters

In this experiment, 40 dedicated nodes from two clusters, i.e. Adonis and Genepi, specified in Table 4.5, of the Grid5000 experimental testbed are used. Each node from Adonis

Table 4.5: Specifications of the GPU-accelerated multicore cluster

Cluster	Adonis (CPU + GPU)		Genepi (CPU)
Processor	Intel Xeon E5520	Nvidia Tesla C1060	Intel XeonE5420 QC
Cores	2×4 cores	240 cores	2×4 cores
Clock Rate	2.27 GHz	602MHz	2.5 GHz
Memory Size	24 GB	4 GB	8 GB
Nodes	9	1 GPU/node	31
Network	Infiniband 40G		Infiniband 20G

cluster is equipped with a NVIDIA GPU, which exacerbates the processor heterogeneity of the experiment platform. In addition, the platform is heterogeneous in terms of memory, because the memory capacity of a Genepi node is much smaller than a Adonis node. We use *gemm* kernels from vendor-optimized BLAS libraries, namely ACML BLAS Library for CPU cores and NVIDIA CUBLAS for GPU. Multi-thread version of ACML *gemm* kernel is used for exploiting the the computing capacity of multicore processors. OpenMPI is used for inter-node communication. All nodes are interconnected by a high speed InfiniBand network which reduces the impact of communication on the total execution time.

In this experiment, multi-threaded ACML and single-GPU computational kernels are used. On each Genepi node, one process executes a multi-threaded computational kernel with eight threads. Therefore, all CPU cores of a Genepi node make a combined processing unit and are modeled by an abstract processor. On each Adonis node, the GPU and its dedicated CPU core make a combined processing unit executing the single-GPU computational kernel, and are modeled by an abstract processor; other seven CPU cores make another combined processing unit executing one multi-threaded computational kernel and are modeled by another abstract processor.

Figure 4.11 shows the pre-built speed functions of CPU and GPU processing units of Adonis and Genepi nodes. The speed function of a Genepi node, $S_8(x)$, is built by executing the multi-threaded computational kernel on the node. As discussed in Section 4.1.3, for *gemm* kernel, it is acceptable to build performance models of a GPU with its dedicated CPU core and other CPU cores separately. Therefore, on a Adonis node, speed functions $g(x)$ and $S_7(x)$ are built separately, since the computational kernel of this application involves mainly matrix-matrix multiplications. When problem sizes are

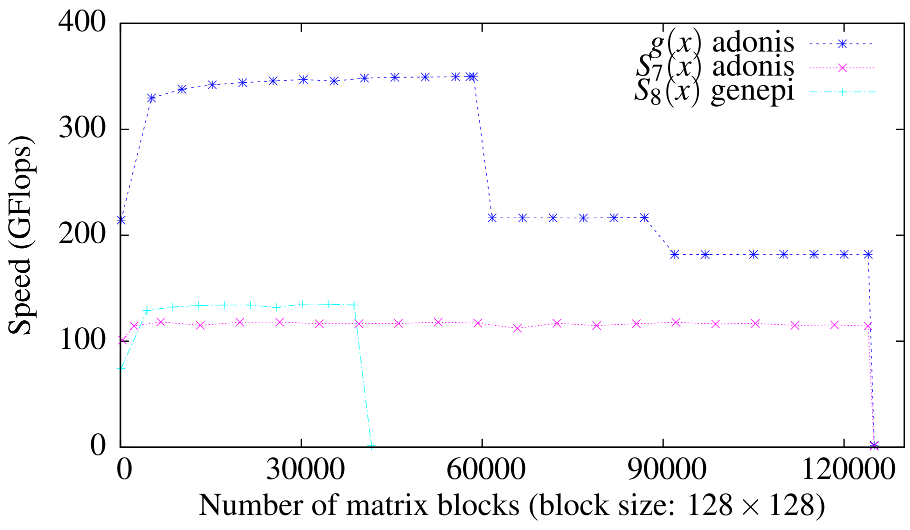


Figure 4.11: Full performance models of Grenoble nodes

small, the relative speed of the three processing elements are constant and as a result data could be partitioned easily; however, when problem sizes exceeded the memory limit of GPU, the performance of the device decreases significantly. In addition, due to a much smaller memory capacity of Genepi nodes, Genepi nodes could only handle relatively small problem sizes. Once problem sizes exceeded the memory limit of a Genepi node, the speed decreased significantly resulting in a great change of relative speed of the three type of processing elements. All these factors complicated the load balancing on such heterogeneous distributed platform.

To demonstrate the accuracy of the partial FPM-based data partitioning on such distributed hybrid platforms, we compared the execution time of the application based on partial FPM-based data partitioning algorithm with that based on CPM-based data partitioning algorithm. Table 4.6 presents the partitionings from the CPM-based and partial FPM-based partitioning algorithms with different problem sizes. Experiments are performed in single precision with blocking factor $b = 128$, and the computation time is presented in seconds. The first column shows the square root of problem sizes. d_1, d_2, and d_3 represent the number of computation units (i.e. matrix blocks of size 128×128) distributed to the GPU processing unit on a Adonis node, the other seven CPU cores on the same Adonis node, and one Genepi node respectively. t_1, t_2, and t_3 represent the computation time of one iteration of the application measured when

Table 4.6: Partitioning results from different data partitioning algorithms on Grenoble

Problem size (blocks)			1100×1100	1200×1200	1300×1300	1400×1400
CPM-based	Adonis GPU	d_1	50000	59700	70200	81800
		t_1	0.60	1.16	1.36	1.57
	Adonis CPU	d_2	16800	19900	23300	27300
		t_2	0.60	0.70	0.83	0.98
	Genepi CPU	d_3	19600	23300	27300	31500
		t_3	0.59	0.72	0.84	0.96
Partial FPM-based	Adonis GPU	d_1	50700	58000	58000	62700
		t_1	0.60	0.68	0.68	1.22
	Adonis CPU	d_2	16900	21100	25700	34000
		t_2	0.59	0.73	0.92	1.24
	Genepi CPU	d_3	19400	23500	30200	34800
		t_3	0.59	0.72	0.91	1.18

computing d_1, d_2, and d_3 computation units on corresponding processing units. The computation time of each iteration is reasonably stable thus it is representative of the total computation time of the application. As we can see, starting from problem size 1200×1200, the CPM-based data partitioning resulted in overloading GPUs due to inaccurate performance models. By contrast, the partially built functional performance model could capture the performance variations, and the data partitioning based on such models was able to balance the load in a wide range of problem sizes.

Figure 4.12 illustrates the execution time (including communication time between processes) of the application when the workload is distributed by different data partitioning algorithms on the hybrid cluster. The execution of the application based on homogeneous partitioning (data distributed evenly) is always unbalanced, being dominated by the slowest processing elements. Both CPM-based and partial FPM-based data partitioning are able to balance the workload when problem sizes are relatively small, i.e. up to 1100×1100. However, starting from problem size 1200×1200, the CPM-based algorithm fails to balance the load and the application takes longer time to complete than that based on the partial FPM-based algorithm. The partial FPM-based data partitioning algorithm reduces the execution time of the application over homogeneous and CPM-based partitioning algorithm by up to 13% and 22% respectively in the range of large problem sizes. Data partitioning based on partial functional performance models achieves comparable load balancing as data partitioning based on pre-build full

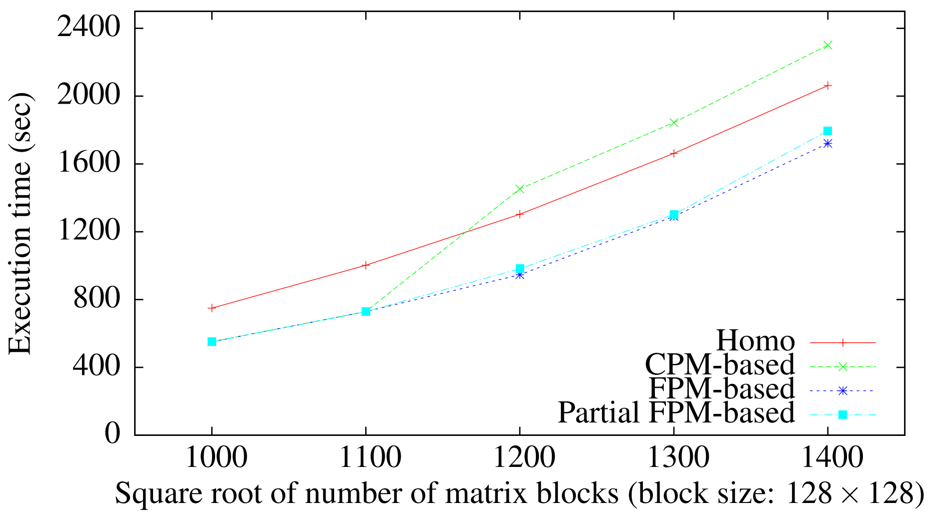


Figure 4.12: Execution time of the parallel matrix multiplication application with different data partitioning algorithms

functional performance models.

Table 4.7 shows the estimated overhead of partial FPM-based data partitioning. The first row shows the problem size, whose square root is the number of the iterations of the application. The second row shows the number of iterations for building partial performance models. The third row shows the ratio of the number of iterations for building partial FPMs to the number of iterations of the application. This ratio could be roughly seen as the ratio of the time for building partial FPMs to the computation time of the application, because the same computational kernel is executed in each iteration of the building of partial FPMs and the execution of the application. As we can see, the time for building performance models is negligible compared to the computation time of the application.

Table 4.7: Estimated overhead of partial FPM-based data partitioning

	1000×1000	1100×1100	1200×1200	1300×1300	1400×1400
Iters	2	2	11	11	9
Ratio	0.2%	0.18%	0.92%	0.85%	0.64%

4.4 Summary

In this chapter, we present performance modeling and performance measurement methods on multicore and multi-GPU systems, and extend the FPM-based data partitioning to heterogeneous multicore and multi-GPU platforms. We also investigate the impact of resource contention on the performance of CPU cores and GPUs, and the impact of NUMA mapping on the performance of GPUs. Using the proposed methods, functional performance models are built and are used as the input of data partitioning algorithms to balance the workload of data parallel applications on target platforms. Experimental results demonstrate that the data partitioning algorithm based on the functional performance models built by the proposed methods is able to balance the workload of data-parallel applications on heterogeneous multicore platforms and achieve good performance.

Chapter 5

Domain Decomposition of Computational Fluid Dynamics Applications on Highly Heterogeneous Modern HPC Platforms

In this chapter, we evaluate the performance modeling and data partitioning methods proposed in previous sections with a computational fluid dynamics application, namely, numerical simulation of lid-driven cavity flow. Based on the functional performance models of the processing units, the geometric domain is divided into a number of subdomians to be processed in parallel. Experimental results prove that the FPM-based data partitioning algorithm is able to balance the load of complex real-life applications on heterogeneous GPU-accelerated multi-core platforms and deliver good performance.

5.1 Introduction to Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is the analysis of systems involving fluid flow, heat transfer, and associated phenomena by means of computer-based numerical simulation [46]. Over the past few decades, computational fluid dynamics has become a practical cornerstone of most fluid and mechanical engineering applications.

Conservation laws describing the motion of fluid flows are derived by considering a certain spatial region in fluid flows, which is called a *control volume* (CV). The fluid motion is governed by the Navier-Stokes equations [46]:

$$\frac{\partial}{\partial t}\int_V \rho \, dV + \int_S \rho \boldsymbol{v} \cdot \boldsymbol{n} \, dS = 0 \tag{5.1}$$

$$\frac{\partial}{\partial t}\int_V \rho \boldsymbol{v} \, dV + \int_S \rho \boldsymbol{v}\boldsymbol{v} \cdot \boldsymbol{n} \, dS = \int_S \mathbf{T} \cdot \boldsymbol{n} \, dS + \int_V \rho \boldsymbol{b} \, dV \tag{5.2}$$

where ρ stands for the density, $\boldsymbol{v}$ for the fluid velocity, t for time, V for the CV volume, S for the CV surface, $\mathbf{T}$ for the stress tensor, and $\boldsymbol{b}$ for the body forces (per unit mass), and $\boldsymbol{n}$ is the unit vector orthogonal to CV surface and directed outwards.

The Navier-Stokes equations are difficult to solve analytically. Therefore, numerical methods are usually used. In numerical methods, the geometric domain and the conservation equations are discretized, producing a system of algebraic equations whose solution is used to approximate the solution of the conservation equations. The system of algebraic equations could be linear or non-linear, depending on the nature of the conservation equations from which they are derived. In the non-linear case, the equations are solved by an iterative technique that involves guessing a solution, linearizing the equations and improving the solution. The process is repeated until a converged result is obtained. In both cases, the linear system of algebraic equations need to be solved efficiently. The coefficient matrices of the linear equations systems are always sparse. The linear equations system can be solved exactly by an *direct* method, such as the Gauss elimination, or approximately by an *iterative* method, such as the conjugate gradient method [51]. Iterative methods are favored when solving large sparse linear systems.

5.2 Parallel Computing in Computational Fluid Dynamics

Parallelization of numerical simulations of fluid dynamics is usually based on domain decomposition, which is essentially data parallelism. In methods based on domain decomposition, the solution domain is divided into a number of subdomains, each assigned to one processor. The problem is solved on the entire domain from problem solutions on subdomains. The same program runs on all processors simultaneously, on its own set of data. Since each processor needs data that resides in other subdomains, exchange of data between processors is necessary. From the viewpoint of partial differential equation, if the value of the solution is known at the interfaces between the subdomains, the equations are decoupled and can be solved concurrently [111]. One example is the *Schur complement method*, which is based on a *non-overlapping* decomposition of the domain. In this method, firstly, a reduced system is derived for determining the values on the subdomain boundaries. Once the reduced problem is solved, the global solution can be obtained by solving a local boundary problem on each subdomain in parallel.

When solving a CFD problem on a parallel platform, it is very important to decide how to map data to processors. Many graph partitioning algorithms have been proposed, which can be used to partitioning the graph representing a system of sparse linear equations. The goal is to subdivide the graph into smaller subgraphs in order to balance the workload among processors and to minimize the amount of communication. Graph partitioning algorithms implemented in Metis [71], Scotch [26], Jostle [125] reduce the number of edges between the target subdomains, aiming to minimize the total communication cost of the parallel application. They take into account the platform heterogeneity, which is specified by a weighted graph providing information about the speed of processors and the bandwidth of links. Algorithms implemented in PaGrid [5], Zoltan [21] minimize the execution time of the application using a cost function, which also depends on the weighted graph of the platform. To distribute data between the processors, all these graph partitioning libraries use simplistic computation performance models, where the speeds of processors are given by constants (weights). Despite the fact that the result of data partitioning is very sensitive to the weights, these libraries do not provide any methods to find the values that balance the workload for given data-parallel applications on heterogeneous platforms.

In [22], a model to predict the execution time of iterative mesh-based applications running on heterogeneous multi-core clusters is proposed. This model takes into account resource heterogeneity, hierarchical communication characteristics, etc., therefore, it can be used to guide the graph partitioning of CFD applications on heterogeneous multi-core environments. Based on the functional performance models of heterogeneous processing units, the data partitioning algorithm [134] can be used to find the accurate weights for a given computational fluid dynamics application on heterogeneous multicore and multi-GPU platforms.

5.3 Model-based Domain Decomposition on Heterogeneous HPC Platforms

In this section, first, we briefly describe the CFD test case and the iterative linear system solvers used to compute its solution. Then, the experimental results on domain decomposition of the CFD test case based on functional performance models on heterogeneous GPU-accelerated multicore platforms are presented.

5.3.1 Test Case: Lid-driven Cavity Flow

The lid-driven cavity flow is a well-known benchmark problem for incompressible, lamina flow of Newtonian fluids. This test case has been studied by many researchers, and accurate solutions are available in the literature (see [48]). The standard case is fluid contained in a square domain with Dirichlet boundary conditions on all sides, with three stationary sides and a lid moving with a tangential unit velocity. The fluid density can be assumed constant for incompressible flows, hence, the Navier-Stokes equations (5.1) and (5.2) reduce to:

$$\int_S \boldsymbol{v} \cdot \boldsymbol{n}\, dS = 0 \tag{5.3}$$

$$\frac{\partial}{\partial t} \int_V \boldsymbol{v}\, dV + \int_S \boldsymbol{v}\boldsymbol{v} \cdot \boldsymbol{n}\, dS = \frac{1}{\rho} \int_S (\mu \nabla \boldsymbol{v} - p) \cdot \boldsymbol{n}\, dS + \int_V \boldsymbol{b}\, dV \tag{5.4}$$

where $\boldsymbol{v}$ is the fluid velocity, t is time, ρ is the density, μ is the kinematic viscosity, V stands for the CV volume, S for the CV surface, and $\boldsymbol{b}$ for the body forces (per unit mass), and $\boldsymbol{n}$ is the unit vector orthogonal to CV surface and directed outwards.

5.3.2 Numerical Solution Methods

For this test case, a system of pressure-velocity coupled equations is produced from discretization of the solution domain and conservation equations. In this work, we use the PISO algorithm [64], an implicit pressure-correction method, to solve this system of algebraic equations.

In PISO, at the beginning of each new time step, the latest solution of velocity and pressure is used as starting estimates. Next, the intermediate velocity field is calculated by solving the linearized momentum equation, and the pressure-correction equations are solved to obtain a pressure-correction value. Then, the velocity field and pressure are corrected and used as new estimates. This process is repeated until all corrections are negligibly small. Finally, the algorithm advances to the next time step. In PISO, the velocity and pressure are calculated by solving linear systems of velocity equations and pressure-correction equations. In this work, the conjugate gradient algorithm (CG) [51] is used to solve the symmetric linear pressure-correction equations. The bi-conjugate gradient stabilized algorithm (BiCGSTAB) [122] is chosen to solve the non-symmetric linear velocity equations.

To solve the CFD test case on parallel platforms, the solution domain is subdivided into a number of subdomains on which the sub-problems are to be solved in parallel. The global iteration matrix is selected so that the diagonal blocks, which contain the elements connecting the nodes that belong to particular subdomains, are decoupled and the subproblems can be solved on subdomains concurrently. After one iteration is performed on each subdomain, the updated values of the unknowns will be exchanged so that the variable values can be corrected and residual can be calculated at nodes near subdomain interfaces.

Algorithm 2 presents the pseudo-code of the CG algorithm, where matrix A is a diagonal block of the global iteration matrix which represents a particular subdomain, ϕ_0 is an initial guess of the result, and matrix M represents the preconditioning matrix and the way of preconditioning. All data arrays, namely matrix A and vectors $\boldsymbol{\phi}$, $\boldsymbol{\rho}$, $\boldsymbol{z}$ and $\boldsymbol{p}$, are local parts for a particular processor operating on a particular subdomain. The main linear algebra operations involved in this solver include matrix-vector products, vector-vector products, and some vector additions and subtractions. Two types of communications are required, namely *local* and *global* communication. Local communication (LC) takes place between processors handling neighboring subdomains. For example, one has to exchange $\boldsymbol{p}_k$ along interfaces, and then update $\boldsymbol{v}_k$ with the product of $\boldsymbol{p}_k$ and corresponding off-diagonal blocks, so that the influence of coupling can be included (which is not shown in the pseudo-code). Global communication (GC) involves all processors. For example, the magnitude of residuals from all processors are gathered and scattered to check if convergence is reached.

Algorithm 3 presents the pseudo-code of the BiCGSTAB algorithm. Similarly, matrix A represents a particular subdomain, ϕ_0 is an initial guess and matrix M represents the preconditioning matrix and the way of preconditioning. All data arrays are local

Algorithm 2: Parallel Conjugate Gradient Algorithm

```
k = 0;
$\rho_0 = Q - A\phi_0$;
while (not convergent) do
    $z_k = M^{-1}\rho_k$;
    $k = k + 1$;
    $s_k = \rho_{k-1}^T z_{k-1}$;
    GC: gather and scatter $s_k$;
    if $k = 1$ then
        $p_1 = z_0$;
    else
        $\beta_k = s_k / s_{k-1}$;
        $p_k = z_{k-1} + \beta_k p_{k-1}$;
    end
    LC: exchange $p_k$ along interfaces;
    $v_k = A p_k$;
    $w_k = p_k^T v_k$;
    GC: gather and scatter $w_k$;
    $\alpha_k = s_k / w_k$;
    $\phi_k = \phi_{k-1} + \alpha_k p_k$;
    $\rho_k = \rho_{k-1} - \alpha_k v_k$;
    GC: gather and scatter $||\rho_k||_2$;
end
```

parts for a particular processor operating on a particular subdomain. The main linear algebra operations involved in this solver include matrix-vector products, vector-vector products, and some vector additions and subtractions. It is worth pointing out that this algorithm requires almost exactly twice as much effort per iteration as the standard conjugate gradient algorithm but converges in about the same number of iterations [46]. Similarly, local communication (LC) that takes place between processors handling neighboring subdomains, and global communication (GC) that involves all processors are also required to calculate scalars such as α, β, ω, and to calculate the magnitude of the global residual to check if convergence is reached.

Algorithm 3: Parallel Bi-Conjugate Gradient Stabilized Algorithm

```
k = 0;
$\boldsymbol{p}_0 = \boldsymbol{\rho}_0 = \boldsymbol{Q} - A\boldsymbol{\phi}_0$;
$\bar{\boldsymbol{\rho}}_0$ is an arbitrary vector, such that $\bar{\boldsymbol{\rho}}_0^T \boldsymbol{\rho}_0 \neq 0$;
while (not convergent) do
    $\hat{\boldsymbol{p}}_k = M^{-1}\boldsymbol{p}_k$;
    LC: exchange $\hat{\boldsymbol{p}}_k$ along interfaces;
    $r_k = \boldsymbol{\rho}_k^T \bar{\boldsymbol{\rho}}_0$;
    $\boldsymbol{v}_k = A\hat{\boldsymbol{p}}_k$;
    GC: gather and scatter $r_k$ and $\boldsymbol{v}_k^T \bar{\boldsymbol{\rho}}_0$;
    $\alpha_k = r_k / \boldsymbol{v}_k^T \bar{\boldsymbol{\rho}}_0$;
    $\boldsymbol{s}_k = \boldsymbol{\rho}_k - \alpha_k \boldsymbol{v}_k$;
    GC: gather and scatter $||\boldsymbol{s}_k||_2$;
    if $||\boldsymbol{s}_k||_2$ is small enough then
        $\boldsymbol{\phi}_{k+1} = \boldsymbol{\phi}_k + \alpha_k \hat{\boldsymbol{p}}_k$;
        exit;
    end
    $\hat{\boldsymbol{s}}_k = M^{-1}\boldsymbol{s}_k$;
    LC: exchange $\hat{\boldsymbol{s}}_k$ along interfaces;
    $\boldsymbol{t}_k = A\hat{\boldsymbol{s}}_k$;
    GC: gather and scatter $\boldsymbol{s}_k^T \boldsymbol{t}_k$ and $\boldsymbol{t}_k^T \boldsymbol{t}_k$;
    $\omega_k = \boldsymbol{s}_k^T \boldsymbol{t}_k / \boldsymbol{t}_k^T \boldsymbol{t}_k$;
    $\boldsymbol{\phi}_{k+1} = \boldsymbol{\phi}_k + \alpha_k \hat{\boldsymbol{p}}_k + \omega_k \hat{\boldsymbol{s}}_k$;
    $\boldsymbol{\rho}_{k+1} = \boldsymbol{s}_k - \omega_k \hat{\boldsymbol{s}}_k$;
    GC: gather and scatter $||\boldsymbol{\rho}_{k+1}||_2$;
    $r_{k+1} = \boldsymbol{\rho}_{k+1}^T \bar{\boldsymbol{\rho}}_0$;
    GC: gather and scatter $r_{k+1}$;
    $\beta_k = (\alpha_k / \omega_k)(r_{k+1} / r_k)$;
    $\boldsymbol{p}_{k+1} = \boldsymbol{\rho}_{k+1} + \beta_k(\boldsymbol{p}_k - \omega_k \boldsymbol{v}_k)$;
    $k = k + 1$;
end
```

5.3.3 Hybrid Linear System Solvers

To take full advantage of hybrid CPU/GPU platforms, we design and implement hybrid CG and BiCGSTAB linear equation solvers. Depending on the type of the computing system on which the hybrid solver is executed, the CPU version or GPU version of code is executed to perform required linear algebra operations in the CG or BiCGSTAB solver. Multiple hybrid solvers can be executed on hybrid CPU/GPU platforms in parallel. Most linear algebra operations involved in the hybrid linear equation solvers are implemented using CUSP 0.3, a C++ template library for sparse matrix computations for both CPU and GPU computing systems.

Parallelism of the hybrid solvers is achieved by employing the single program multiple data paradigm. The local communication (LC) and global communication (GC) between processes that handle subdomains are performed through MPI point-to-point and collective communication routines. For local communication between processes, non-blocking MPI point-to-point communication routines are used to reduce overall communication overhead.

Since CPU and GPU have separate memory spaces, additional data transfers between the main memory and GPU memory are required when running GPU version of the hybrid solvers. The data transfer time between the host and device memory is comparable with the computation time, hence, is included in the measurement of computation performance of GPU solvers.

5.3.4 Experimental Results

In this work, the CFD test case is implemented using the OpenFOAM package [119], a flexible and programmable environment for CFD simulation. Although the OpenFOAM standard solver *icoFoam* can be used to solve the lid-driven cavity problem, it only support parallel computing on CPU platforms. To experiment with heterogeneous GPU-accelerated multicore platforms, we implement the hybrid linear equation solvers described in the preceding section. In OpenFOAM, the pre-processing and post-processing tasks, such as mesh generation and domain decomposition, involve a significant number of file input and output operations which consume a large amount of time. However, these tasks are handled by stand-alone utilities, such as *blockMesh* and *decomposePar* which we use in this study, so the parallelization and load balancing of such tasks are considered separate problems and not studied in this work.

For this simple CFD test case, the square geometric domain is discretized into a regular mesh using the OpenFOAM utility *blockMesh*. In the discretization of the conservation equations, the implicit Euler method is used as the discretization scheme for the first temporal derivative, the Gauss linear method is used for the discretization of the gradient of a given field, and the Gauss linear orthogonal method is used for the discretization of the laplacian of a given field. The discretization scheme for interpolation of values from centers of CVs to face centers is linear interpolation [118]. For the sake of simplicity, the preconditioning in the linear equation solvers is not considered in this work, therefore, the identity matrix is used as the preconditioning matrix.

In the CFD simulation, solving the sparse linear equation systems is the compute-intensive and time-consuming part. Therefore, in order to balance the workload on heterogeneous platforms, we build functional performance models of the processing units executing the linear equation solver, e.g. CG and BiCGSTAB. The speed is calculated by dividing the number of floating-point operations by the computation time of one iteration of the linear equation solver (communication time eliminated). Next, using the FPM-based data partitioning algorithm, we calculate the numbers of control volumes to be assigned to processing units, which are proportional to their speeds. The solution domain is then decomposed into a number of subdomains so that each processing unit handles a subdomain with the number of control volumes assigned to it.

In this work, one-dimensional heterogeneous domain decomposition is used, which is able to handle simple rectangular domains discretized into regular meshes, but the communication is not optimized. It is worth noting that the proposed partitioning method can also be applied to CFD applications with complex geometries which necessitate the use of unstructured meshes. In that case, the numbers of control volumes to be assigned to processing units are used as the input weights of a graph partitioning algorithm, e.g. Metis, to decompose the solution domain, which guarantees a balanced workload and optimized communication overhead on the parallel platform.

According to experimental results, the performance of a processing unit executing the CG and BiCGSTAB solvers is almost the same. The BiCGSTAB solver requires almost exactly twice as many linear algebra operations of each type per iteration as the CG solver. Both the complexity and computation time per iteration of the BiCGSTAB solver are almost twice as much as the CG solver, therefore, their speed remains the same. During the simulation of the lid-driven flow, both the CG and BiCGSTAB solvers will be invoked at each time step. Since the two linear equation solvers have almost the same performance, it is reasonable to partition the workload based on performance models built by executing either solver. In this work, we choose to decompose the domain based on the functional performance models built by executing the CG solver.

The experimental platform is the Adonis cluster, specified in Table 4.5. Each Adonis node consists of two four-core NUMA nodes, and is equipped with a NVIDIA GPU, which exacerbates the processor heterogeneity of the platform. Experiments are conducted on both a single Adonis node and a cluster of Adonis nodes.

In the experiment on a single Adonis node, one GPU CG solver is executed on the GPU and its dedicated CPU core, and one CPU CG solver is executed on each of other seven CPU cores. Therefore, the GPU and its dedicated CPU core make a combined processing unit, and are represented by an abstract processor. Each of other seven cores is modeled by an abstract processor. The processing units of the same NUMA node are grouped together and their speed is measured simultaneously. Their speed is measured for a wide range of problem sizes to build their functional performance models. As a result, three functional performance models are built, which are presented in Figure 5.1. Speed functions $g(x)$ and $s_3(x)$ are built simultaneously by executing the GPU CG solver on the GPU processing unit, and one CPU CG solver on each of other three CPU cores of the same NUMA node. $s_4(x)$ is built by executing one CPU CG solver on each CPU core of the other NUMA node simultaneously. We can see that $g(x)$ is much higher

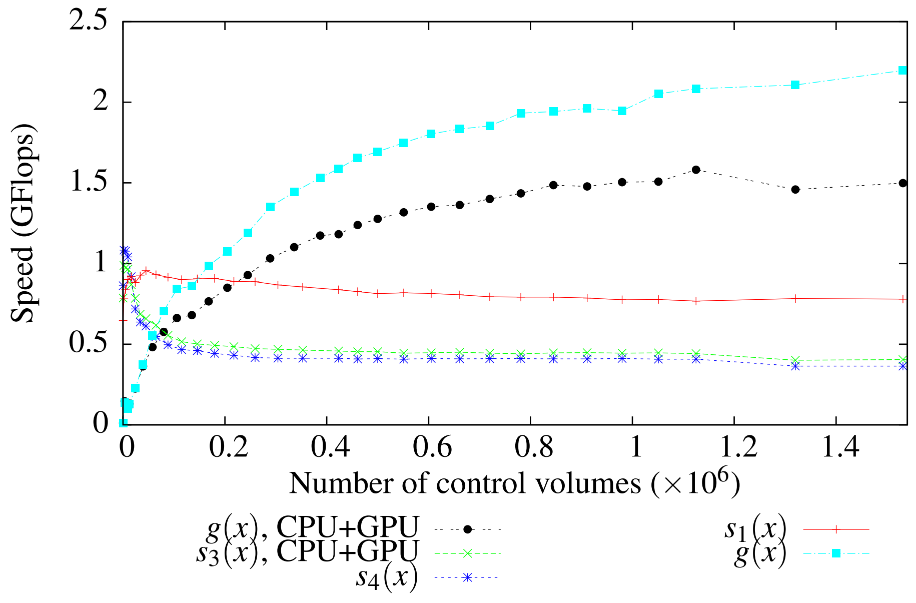


Figure 5.1: Speed functions of the CPU and GPU processing units built by executing the CG solver in different configurations on an Adonis node

than $s_3(x)$ and $s_4(x)$, $s_3(x)$ is slightly higher $s_4(x)$ These functional performance models are then used as the input of FPM-based domain decomposition.

Figure 5.2 presents the speedup in execution time of the CFD test case when using the FPM-based heterogeneous decomposition method over the homogeneous decomposition method, and the estimated upper bound of the actual speedup. It is worth noting that the speedup that can be achieved depends on the level of the processor heterogeneity of the experiment platform.

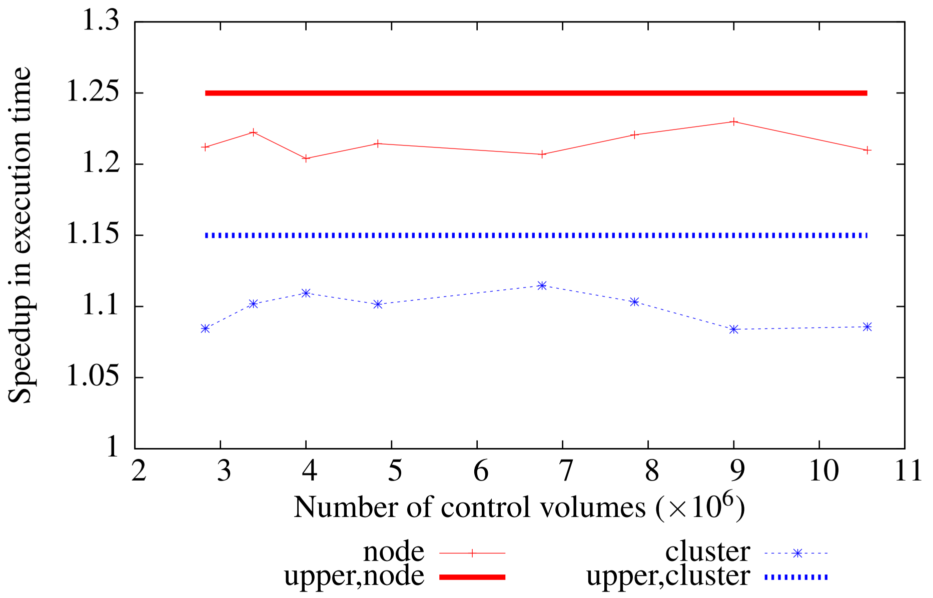


Figure 5.2: The actual speedup and the estimated upper bound of the speedup in execution time on a single Adonis node and on the Adonis cluster

The execution time consists of the computation time and communication time. Therefore, the speedup in execution time, $S_{exec} = (t^h_{comp} + t^h_{comm})/(t^f_{comp} + t^f_{comm})$, where t_{comp} stands for the computation time, t_{comm} stands for the communication time, h indicates that the time is measured in experiments when the homogeneous decomposition is used, and f indicates that FPM-based heterogeneous decomposition is used. Because the geometric domain is decomposed in one-dimension, the total volume of communication does not depend on whether the domain is decomposed homogeneously or heterogeneously. Therefore, t^f_{comm} can be used as an estimate of t^f_{comm}. Divide the numerator and denominator by t^f_{comp}, we get that $S_{exec} = (S_{comp} + r_f)/(1 + r_f)$, where $S_{comp} = t^h_{comp}/t^f_{comp}$, which is the speedup in computation time, $r_f = t^f_{comm}/t^f_{comp}$, which is the ratio of the communication time to the computation time.

In this work, S_{comp} is calculated based on the pre-built functional performance models. To calculate r_f, we measure the execution time, and the communication time with all computation in the linear solvers removed, which can be regarded as a lower bound of the communication overhead. For a given problem size, after the values of S_{comp} and r_f are calculated, the upper bound of the speedup in execution time on the given heterogeneous platform can then be calculated.

As shown in Figure 5.2, the actual speedup in execution time is around 1.22 in experiments on a single Adonis node. For a certain problem size, let g and s_3 denote the speed of the GPU processing unit and the other CPU cores of the same NUMA node, s_4 denotes the speed of the CPU cores of the other NUMA node, with a heterogeneous workload distribution calculated using FPM-based decomposition method. For the same problem size, if the domain is decomposed by homogeneous decomposition method, let $\bar{g}$, $\bar{s}_3$ and $\bar{s}_4$ denote the speed of the corresponding processing units. We can calculate that the speedup in computation time $S_{comp} = (g + 3 * s_3 + 4 * s_4)/(8 * \bar{s}_4)$ is up to 1.4. Based on experimental results, r_f is around 0.6, therefore, the upper bound of S_{exec} is 1.25. As we can see, the actual speedup and the estimated upper bound are quite close, which demonstrates that the FPM-based heterogeneous domain decomposition is able to balance the workload on a GPU-accelerated multicore node.

The mesh generation utility used in the experiments, *blockMesh*, can only run on a single node, so the range of problem size with which we can experiment is limited by the memory capacity of a single node. If the largest number of control volumes that can be generated on a single node are distributed evenly to all processing elements of the Adonis cluster, each will receive only a relatively small number of control volumes. In that case, the computing capacity of GPU is barely fulfilled, and the level of performance heterogeneity of CPU and GPU is low, which is of no interest in this study.

In order to experiment with a wide range of problem sizes on the Adonis cluster, only one GPU (with a dedicated CPU core) and seven CPU cores are used in the experiments, each from a Adonis node. Figure 5.1 shows the speed functions of the CPU and GPU processing units, namely, $s_1(x)$, which is built by executing the CPU CG solver on a single CPU core, and $g(x)$, which is built by executing the GPU CG solver on a GPU processing unit independently. Compared to the performance measured independently, the performance of the GPU processing unit measured under resource contention from neighboring CPU cores reduces by up to 30%.

On the Adonis cluster, the speedup in execution time is around 1.1. Using the same method described above, we can calculate that the speedup in computation time S_{comp} is up to 1.25. Based on experimental results, the ratio of the communication time to the computation time r_f is around 0.67. Therefore, we can estimate that the speedup in execution time S_{exec} is up to 1.15. As we can see, the actual speedup and the estimated upper bound of the actual speedup are reasonably close, which demonstrates the effectiveness of FPM-based heterogeneous domain decomposition.

5.4 Summary

In this chapter, we apply the FPM-based data partitioning to a typical computational fluid dynamics application in order to balance the workload on hybrid GPU-accelerated multicore platforms. First, we briefly introduce the computational fluid dynamics. Next, we briefly review the parallel computing in computational fluid dynamics. Then, using the lid-driven cavity simulation as a test case, we demonstrate that the FPM-based domain decomposition is able to balance the workload of CFD applications on hybrid GPU-accelerated multicore platforms.

Chapter 6

A Software Framework for Data Partitioning on Heterogeneous Platforms

Applying data partitioning for deriving concurrency in parallel scientific applications is challenging in terms of efficient use of heterogeneous hardware and software on the target platform. In order to balance the workload, the computation should be partitioned and distributed to processing elements in proportion to their relative speed. To this end, accurate and efficient benchmarking methods are required to obtain the speed of processing elements; appropriate interpolation methods are required to predict processor performance, and to build computation performance models as input of data partitioning algorithms, such as the functional performance model (Section 2.4.1); data partitioning algorithms (Section 2.4.2) that yield high quality partitioning based on computation performance models should be implemented.

This chapter presents a software framework, Fupermod [35] [1], that meets the challenges of data partitioning for parallel scientific applications on dedicated heterogeneous HPC platforms. The software framework provides functionalities for benchmarking and constructing computation performance models, with the model granularity configurable. Both constant and functional performance models are supported. The performance model can be built either in advance to be used in static data partitioning, or

[1]The software framework was initiated a few years ago in the HCL research group. Many experiments presented in this thesis are conducted with the help of this software framework. The author of the thesis participates in the work on extending the framework to heterogeneous multicore and multi-GPU platforms, including the code refactoring on process configuration, performance measurement, and the development of CBLAS routines for the GPU, the wrappers for memory operations and the execution of linear algebra routines on hybrid GPU-accelerated multicore systems.

at runtime during dynamic load balancing. A set of general-purpose data partitioning algorithms based on computation performance models are also implemented, such as geometrical partitioning algorithm [84] and numerical partitioning algorithm [109]. Moreover, a wide range of dedicated heterogeneous platforms are supported, such as heterogeneous network of uni- or multi-core computers, GPU-accelerated hybrid platforms. The framework is designed to be extensible in that new measurement techniques for new types of hardware can be added and other computation performance models and data partitioning algorithms can be included.

The software framework consists of a library and a set of tools. The library implements the main functionality, which can be integrated into application-level software. The library is made up of five modules: process configuration, performance measurement, model construction, static data partitioning, dynamic data partitioning and load balancing. The tools, which are developed based on the library, are used in different phases of a model-based data partitioning procedure. The framework also includes some user cases such as heterogeneous matrix multiplication and Jacobi solver, and some wrappers of CBLAS routines and memory management for heterogeneous processing elements.

6.1 Library Modules

In this section, we describe the design and implementation of the main five modules of the library of the software framework.

6.1.1 Process Configuration

Configuration settings specific to each MPI process are stored in a single configuration file. Fupermod defines a data structure, *fupermod_process_conf*, to store the configuration information for a specific process; a function, *fupermod_print_conf*, to print a default usable template configuration file of the current mpirun which can be customized further by the user; and a function, *fupermod_get_conf*, to read the configuration file and return the configuration information for that MPI process.

```
typedef struct fupermod_process_conf {
  char* hostname;
  int rank_intra;     // process rank in intra-node communicator
  char* bind;         // processing binding
  char* device_type;
  char* subopts;      // suboptions
} fupermod_process_conf;
```

The data structure *fupermod_process_conf* includes the following parameters: *hostname* specifies the name of the node executing this MPI process; *rank_intra* is the rank of this process in the intra-node MPI communicator; *bind* provides information for process binding; *device_type* specifies the type of the device executing this process (cpu, gpu, etc). Applications designed for hybrid platforms may incorporate codes for different computing devices. In this case, the *device_type* information could be used for choosing the corresponding code for each computing device; *subopts* is a string containing some optional parameters for this process.

```
void fupermod_print_conf( MPI_Comm comm, int root, FILE* file, char*
    default_device_type, char* subopts );
```

The function *fupermod_print_conf* includes the following parameters: *comm* is the MPI communicator; *root* is the rank of the processor to print the template configuration file; *file* is the file pointer to the output template file; *defualt_device_type* specifies the default device type; and *subopts* is a string containing some optional parameters for this process.

```
fupermod_process_conf fupermod_get_conf( MPI_Comm comm, char*
    conf_file_path );
```

The function *fupermod_get_conf* includes two parameters: the MPI communicator *comm* and the path of the configuration file *conf_file_path*. The configuration setting for a specific MPI process contained in a configuration file can be addressed by *hostname* and *rank_intra*. The *hostname* can be obtained by invoking MPI built-in function *MPI_Get_processor_name*. To get the *rank_intra*, we first need to invoke an auxiliary function *fupermod_comm_intra* to get the intra-communicator, then call the MPI built-in function *MPI_Comm_rank*.

```
int fupermod_comm_intra( MPI_Comm comm, MPI_Comm* comm_intra );
int fupermod_bind_process( char* str_mask );
```

The auxiliary function *fupermod_bind_process* can be invoked to bind a process to CPU core(s). It takes a CPU affinity mask as input and then invokes system function *sched_setaffinity* to set this process's CPU affinity mask.

Below is an example of the configuration file. As we can see, 5 MPI processes are configured to be launched, with 3 processes on node *tesla* (8-core) and 1 on node *yeats* (8-core) and 1 on node *hcl02* (dual-core). On node *tesla*, process 0 will be bound to 4 cores from core 0 to core 3, process 1 will be bound to 3 cores from core 4 to core 6, and process 2 will be bound to core 7. Both process 0 and process 1 will run CPU code, and process 2 will run GPU code (offload compute-intensive portions of the application to the GPU). Node *yeats* will execute a single process which further spawn 8 threads utilizing all 8 cores. Similarly, node *hcl02* will execute a single process which further spawns 2 threads utilizing both 2 cores.

```
#hostname     rank_intra  bind  device_type   suboptions
tesla.ucd.ie      0        0-3      CPU     <some optional params>
tesla.ucd.ie      1        4-6      CPU     <some optional params>
tesla.ucd.ie      2        7        GPU     <some optional params>
yeats.ucd.ie      0        all      CPU     OMP_NUM_THREADS=8
hcl02.ucd.ie      0        all      CPU     OMP_NUM_THREADS=2
```

6.1.2 Computation Performance Measurement

Fupermod provide a data structure *fupermod_benchmark* for computation performance measurement, which encapsulates: *kernel*, a pointer to a user-defined benchmark kernel; *conf*, a variable that contains process-specific configuration information; and *execute*, a pointer to a user-defined function that performs the benchmark.

```
typedef struct fupermod_benchmark {
  fupermod_kernel* kernel;
  fupermod_process_conf conf;
  int (*execute) (struct fupermod_benchmark* benchmark, MPI_Comm comm
    , int d, fupermod_precision precision, fupermod_point* point );
} fupermod_benchmark;
```

The data structure *fupermod_kernel* encapsulates user-defined functions required to implement a benchmark: *initialize* and *finalize* point to functions that implement initialization and finalization of memory allocation and deallocation of a problem of *d* computation units, and of the execution context of the kernel based on parameters referenced by pointer *params*; *execute* points to the function that implements the execution of the computational kernel; *complexity* points to the function that returns the complexity of computing *d* computation units. Please note that the code of the (serial) computational kernel has to be provided by the user along with a function that converts speed from units/sec to floating-point performance.

```
typedef struct fupermod_kernel {
  int (*initialize)(int d, void* params);
  int (*execute)(pthread_mutex_t* mutex, void* params);
  int (*finalize)(void* params);
  double (*complexity)(int d, void* params);
} fupermod_kernel;
```

During the performance measurement, function *execute* of *fupermod_benchmark* executes the computational kernel encapsulated in *fupermod_kernel* multiple times to obtain an reliable result. The *precision* argument specifies how many times to repeat the benchmark. User can define the minimum and maximum numbers of repetitions. The benchmark will stop when the number of repetitions has reached *reps_max*. Otherwise it will be repeated until the sample of the measured execution times satisfies the Student's t-test with the confidence level *cl* and relative error *eps*. The measured execution time is proved reliable statistically if $ci/\bar{T} < eps$, where *ci* is the confidence interval that contains the average execution time with a certain probability $Pr(|\bar{T} - \mu|) = cl$, $\bar{T}$ is the average time obtained from several observations, and μ is the mean. An auxiliary function *fupermod_ci* is provided to return the confidence interval.

```
typedef struct fupermod_precision {
  int reps_min; // minimum number of repetitions
  int reps_max; // maximun number of repetitions
  double cl;    // confidence level
  double eps;   // relative error
} fupermod_precision;
double fupermod_ci(double cl, int reps, double* t);
```

The measurement result is returned in the form of an experimental *point* of data type *fupermod_point*, which contains information on problem size and execution time. Auxiliary functions are provided to output experimental points to files and to read them from a file. Such experimental points are used as the input of functions for model construction to build computation performance models.

```
typedef struct fupermod_point {
  int d;      // number of computation units
  double t;   // measured execution time
  int reps;   // number of repetitions
  double ci; // confidence interval
} fupermod_point;
```

The following code shows an implementation of the benchmark function *execute* of *fupermod_benchmark* for the purpose of demonstration. The basic idea is that it invokes functions encapsulated in data structure *fupermod_kernel* to initialize, finalize the computational kernel, and execute the computational kernel multiple times to obtain an accurate benchmark result.

```
void benchmark_execute(fupermod_benchmark* benchmark, MPI_Comm
    comm, int d, fupermod_precision precision, fupermod_point*
    point) {
  // initialize
  void* params;
  benchmark->kernel->initialize(d, &params);
  // execution
  int stop = 0; // loop termination condition
  int reps = 0; // number of repetitions
  double sum = 0; double ci = 0;
  double* t = (double*)malloc(sizeof(double) * precision.reps_max)
   ;
  while (reps < precision.reps_max && !stop) {
    int accurate = 0;
    MPI_Barrier(comm);
    struct timespec start, end;
    get_time(&start);
    benchmark->kernel->execute(NULL, params);
    get_time(&end);
    double time = get_timediff(&start, &end);
    sum += t[reps] = time;
    ci = fupermod_ci(precision.cl,reps,t);
    if (reps>=precision.reps_min-1 && ci*reps/sum < precision.eps)
```

```
      accurate = 1;
    MPI_Allreduce(&accurate, &stop, 1, MPI_INT, MPI_MIN, comm);
    reps++;
  } free(t);
  // result
  point->d = d;
  point->t = sum/reps;
  point->reps = reps;
  point->ci = ci;
  // finalize
  benchmark->kernel->finalize(params);
}
```

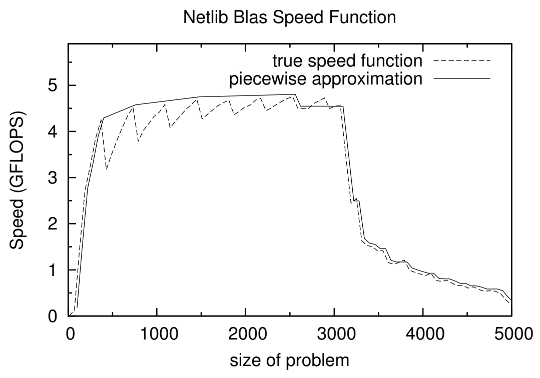
6.1.3 Performance Model Construction

The key abstraction of the programming interface for performance model construction is *fupermod_model*, which has the following interface: *data*, a variable of *fupermod_data* data type that encapsulates experimental points obtained from measurements under different problem sizes; *t*, a user-defined function that implements the approximation of execution time; *update*, a user-defined function that specifies how the approximation changes after adding a new experimental point. These approximations are used in the model-based data partitioning algorithms to predict the computation performance and distribute the workload proportionally.

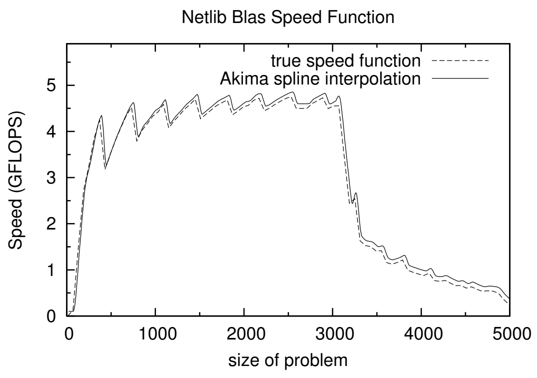
```
typedef struct fupermod_model {
  fupermod_data* data;     // experimental points
  double (*t)(struct fupermod_model* model, double x);
  int (*update)(struct fupermod_model* model, fupermod_point point);
} fupermod_model;

typedef struct fupermod_data {
  int count;               // Number of points
  fupermod_point* points; // points
} fupermod_data;
```

Currently, Fupermod supports both the constant performance model and the functional performance model. Two types of interpolation techniques are implemented to construct the execution time function, namely, piecewise linear interpolation and Akima spline interpolation.

The functional performance model constructed using linear piecewise interpolation is based on some assumptions on the shape of the speed function [84], where data points that do not satisfy the assumptions are removed, as shown in Figure 6.1(a). The functional performance model based on the Akima spline interpolation removes these restrictions [109], and therefore, represents the speed of the processor with more accurate continuous functions (Figure 6.1(b)). The *fupermod_model* data structure can be used to implement other computation performance models.



(a)



(b)

Figure 6.1: Speed functions of the matrix multiplication kernel based on the Netlib BLAS GEMM: (a) piecewise linear interpolation (b) Akima spline interpolation

6.1.4 Static Data Partitioning

Fupermod defines the following programming interface for static data partitioning algorithms, where *size* is the number of the processes running the parallel application, *models* is an array of performance models of the processes, and *dist* specifies how workload (data) will be distributed among these processes.

```
typedef int (*fupermod_partition)( int size, fupermod_model** models,
    fupermod_dist* dist );
```

The user should provide computation performance models of processes as the input of data partitioning algorithms. After execution of the data partitioning algorithm, the workload distribution result is returned and stored in a *fupermod_dist* variable. Auxiliary functions for outputting the distribution result to a file and for reading in distribution information from a file are provided. The application programmer can then distribute data to processes based on the distribution result generated by the data partitioning algorithm.

The data structure for storing information on data distribution is *fupermod_dist*. It has the following parameters: D is the total problem size to partition (in computation units); *size* is the number of processes; *parts* is an array of partitions each specifying the workload d to be assigned to a process i and the predicted computing time t of the workload.

```
typedef struct fupermod_dist {
  int D;                   // total problem size
  int size;                // number of processes
  fupermod_part* parts;    // partitions
} fupermod_dist;
typedef struct fupermod_part {
  int d;                   // partitioned problem size
  int i;                   // process index
  double t;                // predicted execution time
} fupermod_part;
```

Currently, three data partitioning algorithms have been implemented:

- *fupermod_partition_constant* implements a basic partitioning algorithm based on constant performance models. This algorithm divides the data in proportion to the speeds represented by constants.

- *fupermod_partition_geometric* implements a geometrical partitioning algorithm [84] based on functional performance models constructed by piecewise linear interpolation. The algorithm implements iterative bisection of the speed functions with lines passing through the origin of the coordinate system. Convergence of this algorithm is ensured by putting restrictions on the shape of the speed functions (Section 2.4.2).

- *fupermod_partition_multiroot* implements a numerical partitioning algorithm based on functional performance models constructed by Akima spline interpolation. The algorithm applies multidimensional solvers to numerical solution of the system of non-linear equations that formalize the problem of optimal data partitioning [109]. It can be applied to smooth speed functions of any shape.

6.1.5 Dynamic Data Partitioning and Load Balancing

Fupermod provides efficient data partitioning algorithms that do not require performance models as input but partially estimate the performance model at runtime to a sufficient level of accuracy (Section 2.4.2). These algorithms allow for efficient load balancing and are suitable for use in self-adaptable applications. The dynamic partitioning algorithms build the partial functional performance models iteratively and alternate between (i) performing benchmark on each process for a given distribution of workload, adding newly measured speeds to the corresponding partial models and (ii) repartitioning the data based on current partial estimates of the speed functions. In the case of dynamic data partitioning, the measurements are made by benchmarking the representative computation kernel of the application. In the case of dynamic load balancing, the execution of one iteration of the application is timed.

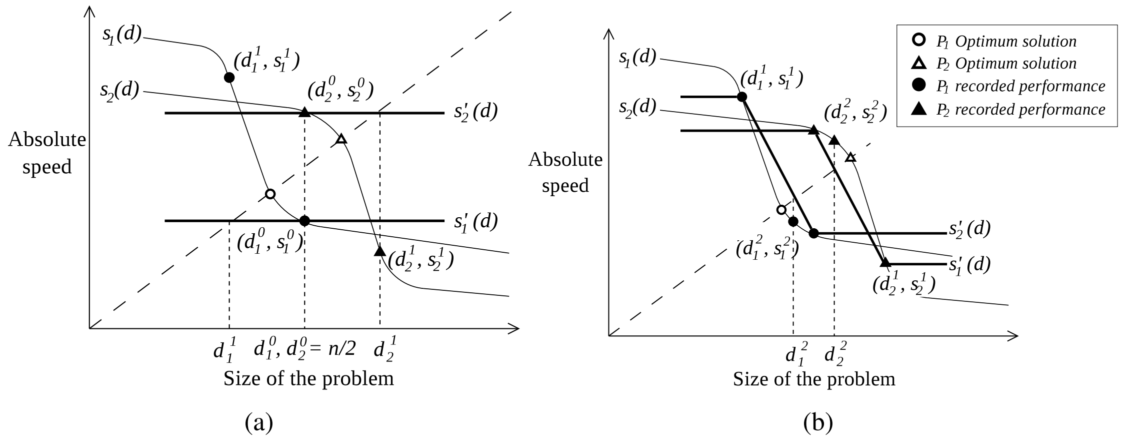


Figure 6.2: Construction of the partial FPMs based on piecewise linear interpolation, using the geometrical data partitioning algorithm

Data structure *fupermod_dynamic* is defined to encapsulate execution context required for dynamic data partitioning or load balancing algorithms. Parameter *partition* specifies a static data partitioning algorithm which will be used repeatedly in the dynamic algorithm; *size* is the number of processes participating in the dynamic algorithm;

models stores current partial functional performance models which is updated iteratively during the dynamic algorithm; *dist* stores the current distribution. If current distribution *dist* is well balanced, the dynamic algorithm stops repartitioning; otherwise, the dynamic algorithm repeats another benchmark using *partition* algorithm based on current partial models *models*.

```
typedef struct fupermod_dynamic {
  fupermod_partition partition;
  int size;
  fupermod_model** models;
  fupermod_dist* dist;
} fupermod_dynamic;
```

Function *fupermod_partitioner_iterate* implements one step of the dynamic data partitioning algorithm. First, it distributes workload to processes according to the data distribution stored in *partitioner*; next, it performs one benchmark with computational kernel encapsulated in parameter *benchmark*; then, it gathers newly measured speeds and add them to current *models*; finally, it repartition the workload and compare the distribution obtained from repartition with current distribution *dist*. If the distribution change is within relative error *eps*, status indicating convergence is returned; otherwise, status indicating another iteration is returned.

```
int fupermod_partitioner_iterate( fupermod_dynamic* partitioner,
    MPI_Comm comm, fupermod_precision precision, fupermod_benchmark*
    benchmark, double eps );
```

The code snippet below demonstrates dynamic data partitioning.

```
fupermod_dynamic partitioner;
fupermod_benchmark* benchmark = fupermod_benchmark_alloc();
// status: 0 indicates more iterations, 1 indicates convergence
int status = 0;
while (!status) {
  fupermod_partitioner_iterate(&partitioner, comm, root,
   precision, benchmark);
  if (rank == root)
    status = fupermod_dist_test(partitioner.dist, eps);
  MPI_Bcast(&status, 1, MPI_INT, root, comm);
}
fupermod_benchmark_free(benchmark);
```

Function *fupermod_balancer_iterate* implements one step of the dynamic load balancing algorithm. It is invoked after the execution of each iteration of the application. First it measures the execution time of each iteration, next, it adds the measured data to update partial functional performance models, then, it repartition the workload based on updated performance models.

```
int fupermod_balancer_iterate( fupermod_dynamic* partitioner,
    MPI_Comm comm, struct timespec start);
```

The code snippet below demonstrates dynamic load balancing of a data-parallel application Jacobi solver. This application distributes the matrix and vectors by rows between the processors and iteratively solves the system of equations. In the code snippet below, the partial FPMs based on piecewise linear interpolation are constructed at runtime during the iterations of the Jacobi method. At each iteration, the load balancing function invokes the geometrical data partitioning algorithm. The system of equations is redistributed accordingly to the newly obtained data distribution.

```
MPI_Comm_size(comm, &size);
// FPMs based on piecewise linear interpolation
fupermod_model** models = malloc(sizeof(fupermod_model*) * size);
for (i = 0; i < size; i++)
  models[i] = fupermod_model_piecewise_alloc();
// context for dynamic load balancing
fupermod_dynamic balancer = { fupermod_partition_geometric, size,
    models, fupermod_dist_alloc(D, size) };
// current distribution, initially even
fupermod_dist* dist = fupermod_dist_alloc(D, size);
// Jacobi data: dist->parts[i].d rows of matrix and vectors
double *A, *b, *x; // allocation, initialization
// main loop
double error = DBL_MAX;
while (error > eps) {
  // redistribution of Jacobi data accordingly to balancer.dist
  jacobi_redistribute(comm, dist, A, b, x, balancer.dist);
  // store the current distribution
  fupermod_dist_copy(dist, balancer.dist);
  struct timeval start;
  gettimeofday(&start, NULL);
  // Jacobi iteration
  jacobi_iterate(comm, dist, A, b, x, &error);
  // load balancing with the (dist->parts[i].d, now-start) point
  fupermod_balance_iterate(&balancer, comm, start);
}
```

6.2 Wrappers

In order to facilitate programming hybrid CPU+GPU platforms, Fupermod defines wappers for memory operations and one of the level 3 CBLAS routines, i.e. general matrix multiplication. More wrappers for CBLAS routines could be implemented in the future work.

Two memory function wrappers are defined as below. Function *fupermod_malloc* is designed to allocate memory of size specified in parameter *size*. It invokes CPU memory operation *malloc* or GPU memory operation *cudaHostAlloc* depending on the device type stored in process-specific configuration *conf*. Additional parameters such as memory allocation modes, device id, can be passed to the function through field *subops* of configuration *conf*. Function *fupermod_free* is designed to free memory block referenced by pointer *ptr*. It invokes CPU memory operation *free* or GPU memory operation *cudaFreeHost* according to device type from *conf*. Additional parameters required for memory free are passed to the function through *subops* that is a field of configuration *conf*.

```
void* fupermod_malloc(size_t size, fupermod_process_conf* conf);
void fupermod_free(void* ptr, fupermod_process_conf* conf)
```

Currently, Fupermod implements a wrapper function *fupermod_gemm_execute* for general matrix multiplication on different computing devices. Data structure *fuper-*

mod_gemm encapsulates process-specific configuration *conf* and additional parameters *params* for the *gemm* routine. Auxiliary functions, *fupermod_gemm_alloc* and *fupermod_gemm_free* are provided for initialization and finalization of execution context of gemm routine.

```
void fupermod_gemm_execute( fupermod_gemm* gemm, const enum
    CBLAS_ORDER Order, const enum CBLAS_TRANSPOSE TransA, const enum
    CBLAS_TRANSPOSE TransB, const int M, const int N, const int K,
    const fupermod_float alpha, const fupermod_float *A, const int lda
    , const fupermod_float *B, const int ldb,  const fupermod_float
    beta, fupermod_float *C, const int ldc );

typedef struct fupermod_gemm {
  fupermod_process_conf* conf; // configuration
  void* params;                // additional parameters for gemm
    routine
} fupermod_gemm;

fupermod_gemm* fupermod_gemm_alloc(fupermod_process_conf* conf);
void fupermod_gemm_free(fupermod_gemm* gemm);
```

6.3 Tools

In this section, we first describe a set of tools that have been implemented in Fupermod to support all steps involved in a model-based data partitioning, then we give a quick start guide on how to use these tools to conduct a data partitioning.

The *builder* is a parallel MPI-based executable program that performs a speed benchmark using a user-defined computational kernel, and outputs speed measurements for a series of problem sizes into a data file. This file contains all data required for construction of computation performance models. Some of its command-line arguments are shown below. The user-defined computational kernel is implemented as a shared library, then *builder* can link against this library to invoke the computational kernel for benchmark. User can also specify path to output data file, a range of problem sizes, number of benchmark steps, and so on.

```
-l S  shared library (computational kernel)
-f S  path to output data file (default: %s)
-L I  lower problem size (default:%d)
-U I  upper problem size (default:%d)
-s I  number of steps in the model (default %d)
-r I  minimum number of repetitions (default %d)
-R I  maximum number of repetitions (default %d)
-i D  confidence level (default %f)
-e D  relative error (default %f)
```

The *modeler* is a serial tool to test the approximation of kernel's time and speed, using different computation performance models and different methods of interpolation and smoothing.

The *partitioner* a serial tool that performs model-based data partitioning. It offers a variety of choices of data partitioning algorithms. It takes the output data files generated by Tool *builder* as input, and outputs a data file containing optimal distribution. The application should take this output file as input to distribute workload. Some of its command-line arguments are shown below.

```
-a I  Balancing algorithm. (default: %d)
   0: Homogeneous distribution
   1: Constant performance model 1 (small benchmark)
   2: Constant performance model 2 (homogeneous benchmark)
   3: Geometric algorithm
   4: multiroot solver
-l S shared library (computational kernel)
-D I problem size to partition (required)
-s I smoothing, number of points to average over(default: %d)
-f S path to input data file (default: %s)
-m S path to machine file (default: %s)
-p S path to output distribution (default: %s)
```

The *dynamic_partitioner* is a parallel MPI-based tool that performs dynamic data partitioning. It builds partial functional performance models at runtime and output a data file containing near-optimal distribution. Some of the command-line arguments are shown below.

```
-a I  Balancing algorithm. (default: %d)
  0: Homogeneous distribution
  1: Constant performance model 1 (small benchmark)
  2: Constant performance model 2 (homogeneous benchmark)
  3: Geometric algorithm
  4: FPM multiroot solver
-l S  shared library (computational kernel)
-p S  path to output inter-node distribution (default: %s)
-d I  problem size to partition (default %lld)
-i I  number of inter-node iterations (default: %d)
-e I  Stopping condition eps (default: %lf)
```

Below is a quick start guide on how to use these tools to generate optimal distribution to balance workload of data-parallel applications on heterogeneous platforms. In step 1, data files used as input to construct performance models are generated by executing *builder* in parallel. Application executed in Step 3 should take data file *partition.dist* generated in step 2 as input to distribute workload.

```
Step 1: Run the builder in parallel
  $ mpirun -np 8 -hostfile hostfile builder -l <install dir>/libs/
   lib_kernel.so -U2000 -s100
  // data file used as input to construct models
  Output file: hostname.rank.device_type.fpm

Step 2: Run partitioner on head node
  $ partitioner -l <install dir>/libs/lib_kernel.so -a3 -D256
  // data file containing optima- distribution
  Output file: partition.dist

Step 3: Execute application
 $ mpirun -n 8 -hostfile hostfile application_excutable
```

6.4 Summary

In this chapter, we outline a software framework for general-purpose data partitioning based on computation performance models. This software framework provides a range of algorithms and models for optimization of data-parallel scientific applications on modern heterogeneous platforms.

Chapter 7

Conclusion

The design of microprocessor has been shifting to a new model where multiple homogeneous processing units, aka cores, are integrated onto the same die due to the heat dissipation and energy consumption issues. Multicore clusters have become the mainstream HPC platforms, which are naturally heterogeneous in terms of processor performance and memory capacity. In the meantime, specialized processing resources, such as Graphic Processing Units (GPUs), has been widely incorporated into multicore computing systems for gaining extra computing power, thus making a computing system heterogeneous. This thesis focuses on the problem of optimal data distribution of data parallel applications on heterogeneous multicore and heterogeneous GPU-accelerated multicore platforms. In the thesis, we provide solutions to the performance modeling and data partitioning on target heterogeneous platforms, and present a software framework for facilitating performance modeling and data partitioning on target platforms.

In Chapter 3, we present the performance modeling and performance measurement methods of multicore systems, and extend the FPM-based data partitioning to heterogeneous multicore clusters. The performance modeling of multicore systems is complicated by resource contention between CPU cores. We model a multicore system by a number of abstract processors determined by the configuration of the parallel application. Each abstract processor represents a CPU processing unit made of one or a group of CPU cores executing one computational kernel of the application. For example, if a single-threaded computational kernel is used, then each CPU core will be modeled by an abstract processor. If a multi-threaded computational kernel is used, then each group of CPU cores executing the kernel makes a combined processing unit, and will be modeled by an abstract processor. To measure the performance accurately, we propose to group CPU processing units by shared resources, so that some system resources are shared within each group but not between groups. The performance of CPU processing units in a group is measured when all the CPU processing units are executing some workload simultaneously, thereby taking into account the influence of resource contention. Using proposed method for measuring performance, functional performance models of CPU processing units are built, and used as the input of data partitioning algorithm to balance the workload on target platforms. Experimental results demonstrate that data partitioning algorithms based on functional performance models built by proposed methods are able to balance the workload of data parallel applications on target platforms and deliver good performance.

In Chapter 4, we present performance modeling and performance measurement methods of multicore and multi-GPU systems, and extend the FPM-data partition to heterogeneous multicore and multi-GPU platforms. We propose to model a multicore and multi-GPU system by a number of heterogeneous abstract processors determined by the configuration of the parallel application. Each abstract processor represents a processing unit made of one or a group of processing elements executing one computational kernel of the application. The method of performance modeling of CPU cores is the same as in Chapter 3. For GPUs, if a single-GPU computational kernel is used, then the GPU and its dedicated CPU core make a combined processing unit, and will be modeled by an abstract processor. If a multi-GPU computational kernel is used, then the GPUs and their dedicated CPU core make a combined processing unit, and will be modeled by an abstract processor. To measure the performance accurately, we propose to group processing units (including both CPU and GPU processing units) by shared resources, so that some system resources are shared within each group but not between groups. The performance of processing units in a group is measured when all processing units in the group are executing some workload simultaneously. We investigate the impact of resource contention on the performance of CPU and GPU processing units, and the impact of processing mapping on GPU-accelerated multicore systems of NUMA architecture on the performance of the GPU processing unit. We build functional performance models of abstract processors, and partition data using the functional performance models to balance the load between heterogeneous processing units. Experimental results with a parallel matrix multiplication application on both a single hybrid server and on a hybrid cluster demonstrate that data partitioning algorithms based on proposed performance modeling method are able to balance the workload of data parallel applications on target platforms and deliver good performance.

In Chapter 5, we evaluate the proposed performance modeling and data partitioning methods with a typical computational fluid dynamic application, namely numerical simulation of lid-driven cavity flow. The fluid motion is governed by the Navier-Stoke equations, which are different to solve analytically, so numerical method need to be used. Through discretization of the geometric domain and the conservation equations, a system of algebraic equations is formed, whose solution can be used to approximate the solution of the conservation equations. Parallel computing in computational fluid dynamics is usually based on domain decomposition, which is essentially data parallelism. In domain decomposition methods, the geometry domain is divided into a number of subdomains, each assigned to a process. The problem is solved on the entire domain from problem solutions on subdomains. To balance the load between subdomains, we build functional performance models of platform's processing units executing the linear equation solvers used for the solution of the test case, i.e. conjugate gradient and biconjugate gradient stabilized algorithms, for solving symmetric pressure equations and non-symmetric velocity equations respectively. Based on the functional performance models, the domain is divided into a number of subdomians in proportion to the speed of processing units handling the subdomains. Experimental results on both hybrid server and cluster prove that FPM-data partitioning algorithms are able to balance the workload of complex applications and deliver good performance.

In Chapter 6, we outline a software framework that is designed to facilitate the performance modeling and data partitioning on heterogeneous platforms. The software framework is made up of a library and a number of standalone tools. The library provides function interfaces for process configuration, performance measurement, model construction, static data partitioning, dynamic data partitioning and load balancing. Snippet code is shown to demonstrate how to dynamically balance the workload of a data-parallel application Jacobi solver. The tools, which are developed based on the library, can be used in different stages of a model-based data partitioning procedure. A quick start guide on how to use these tools is presented at the end of this chapter.

The results presented in Chapter 3 have been published in [133], part of the results presented in Chapter 4 have been published in [134], the results presented in Chapter

6 have been published in [35], and the results presented in Chapters 4, 5 have been submitted for publication in IEEE Transactions on Computers.

As high performance computing systems evolve, the requirement for balancing workload between diverse heterogeneous processing elements will become stronger and challenging. For data-parallel scientific applications, the data partitioning approach based on functional performance models built using proposed methods is an efficient solution that can reliably solve this problem.

Bibliography

[1] Advanced Micro Devices, Inc. (AMD). AMD Core Math Library. `http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml`.

[2] Advanced Micro Devices, Inc. (AMD). clBLAS. `https://github.com/clMathLibraries/clBLAS`.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[4] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Supercomputing '90., Proceedings of*, pages 2–11, Nov 1990.

[5] E. Aubanel and X. Wu. Incorporating latency in heterogeneous graph partitioning. In *IPDPS 2007*, pages 1–8, 2007.

[6] Cedric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *EuroPar'09*, pages 56–65, 2009.

[7] Cedric Augonnet, Samuel Thibault, Raymond Namyst, et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *EuroPar'09*, pages 863–874, 2009.

[8] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM.

[9] S. Barrachina, M. Castillo, F.D. Igual, et al. Evaluation and tuning of the Level 3 CUBLAS for graphics processors. In *IPDPS 2008*, pages 1–8, 2008.

[10] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix-matrix multiplication on heterogeneous platforms. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 289–298, 2000.

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

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

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

[14] Siegfried Benkner et al. High-level support for pipeline parallelism on many-core architectures. Euro-Par'12, pages 614–625, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[16] Javier Garcia Blas, Florin Isaila, Monica Abella, Jesus Carretero, Ernesto Liria, and Manuel Desco. Parallel implementation of a x-ray tomography reconstruction algorithm based on mpi and cuda. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 217–222, New York, NY, USA, 2013. ACM.

[17] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28–37, 2012.

[18] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, January 2010.

[19] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[20] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[21] U.V. Catalyurek, E.G. Boman, K.D. Devine, et al. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *IPDPS 2007*, pages 1 –11, 2007.

[22] S.Y. Chan, T.C. Ling, and E. Aubanel. Performance modeling for hierarchical graph partitioning in heterogeneous multi-core environment. *Parallel Computing*, 2014.

[23] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[24] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In

Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on, pages 44–54, 2009.

[25] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with gpus and fpgas. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 101–107, 2008.

[26] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, July 2008.

[27] Jaeyoung Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *HPC Asia '97*, pages 224 –229, 1997.

[28] Jaeyoung Choi, J.J. Dongarra, R. Pozo, and D.W. Walker. Scalapack: a scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127, Oct 1992.

[29] D. Clarke. Design and implementation of parallel algorithms for modern heterogeneous platforms based on the functional performance model. Phd thesis, University College Dublin, Dublin, 05/2014 2014.

[30] D. Clarke, A. Ilic, A. Lastovetsky, V. Rychkov, L. Sousa, and Z. Zhong. *Design and optimization of scientific applications for highly heterogeneous and hierarchical HPC platforms using functional computation performance models*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2013.

[31] D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa. Hierarchical partitioning algorithm for scientific computing on highly heterogeneous cpu + gpu clusters. In *18th International European Conference on Parallel and Distributed Computing (Euro-Par 2012)*, pages 489–501, Rhodes Island, Greece, 27-31 August 2012. Lecture Notes in Computer Science 7484, Springer, Lecture Notes in Computer Science 7484, Springer.

[32] D. Clarke, A. Lastovetsky, and V. Rychkov. Dynamic load balancing of parallel computational iterative routines on platforms with memory heterogeneity. In *Europar 2010 / Heteropar'2010*, pages 41–50, Ischia-Naples, Italy, 31/08/2010

2011. Lecture Notes in Computer Science 6586, Springer, Lecture Notes in Computer Science 6586, Springer.

[33] D. Clarke, A. Lastovetsky, and V. Rychkov. Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models. In *9th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2011)*, pages 450–459, Bordeaux, France, August 29, 2011 2012. Lecture Notes in Computer Science 7155, Springer, Lecture Notes in Computer Science 7155, Springer.

[34] D. Clarke, Z. Zhong, V. Rychkov, and A. Lastovetsky. Fupermod: a framework for optimal data partitioning for parallel scientific applications on dedicated heterogeneous hpc platforms. In *12th International Conference on Parallel Computing Technologies (PaCT-2013)*, pages 182–196, St. Petersburg, Russia, 30 Sept - 4 Oct 2013. Lecture Notes in Computer Science 7979, Springer, Lecture Notes in Computer Science 7979, Springer.

[35] David Clarke, Ziming Zhong, V. Rychkov, and Alexey Lastovetsky. Fupermod: a software tool for the optimization of data-parallel applications on heterogeneous platforms. *The Journal of Supercomputing*, 69:61– 69, 2014.

[36] Phyllis E. Crandall, Michael J. Quinn, Contact Dr, and Phyllis E. Cr. Problem decomposition for non-uniformity and processor heterogeneity. *Journal of the Brazilian Computer Society*, 2:13–23, 1995.

[37] M. Daga, A.M. Aji, and Wu chun Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141–149, 2011.

[38] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an apu. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 240–247, 2012.

[39] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop*

on General-Purpose Computation on Graphics Processing Units, GPGPU '10, pages 63–74, New York, NY, USA, 2010. ACM.

[40] Michael C. Delorme, Tarek S. Abdelrahman, and Chengyan Zhao. Parallel radix sort on the amd fusion accelerated processing unit. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 339–348, 2013.

[41] Sérgio Dias and Abel Gomes. Triangulating molecular surfaces on multiple gpus. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 181–186, New York, NY, USA, 2013. ACM.

[42] J. Dongarra. Trends in high performance computing: a historical overview and examination of future developments. *Circuits and Devices Magazine, IEEE*, 22(1):22–27, Jan 2006.

[43] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[44] E. Dovolnov, A. Kalinov, and S. Klimov. Natural block data decomposition for heterogeneous clusters. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, 2003.

[45] A.E. Eichenberger, K. O'Brien, K. O'Brien, Peng Wu, Tong Chen, P.H. Oden, D.A. Prener, J.C. Shepherd, Byoungro So, Z. Sura, A. Wang, Tao Zhang, Peng Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 161–172, 2005.

[46] Joel H. Ferziger and Milovan Peric. *Computational Methods for Fluid Dynamics*. Springer, 3rd edition, 2002.

[47] I. Galindo, F. Almeida, and J. Bada-Contelles. Dynamic Load Balancing on Dedicated Heterogeneous Systems. In *EuroPVM/MPI 2008*, pages 64–74. Springer, 2008.

[48] U. Ghia, K. N. Ghia, and C. T. Shin. High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method. *Journal of Computational Physics*, 48:387–411, December 1982.

[49] R. Giladi and N. Ahitav. Spec as a performance evaluation measure. *Computer*, 28(8):33–42, 1995.

[50] G. Goldrian, T. Huth, B. Krill, J. Lauritsen, H. Schick, I. Ouda, S. Heybrock, D. Hierl, T. Maurer, N. Meyer, A. Schafer, S. Solbrig, T. Streuer, T. Wettig, D. Pleiter, K.-H. Sulanke, F. Winter, H. Simma, S.F. Schifano, and R. Tripiccione. Qpace: Quantum chromodynamics parallel computing on the cell broadband engine. *Computing in Science Engineering*, 10(6):46–54, 2008.

[51] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[52] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[53] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35:4:1–4:14, July 2008.

[54] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *Micro, IEEE*, 26(2):10–24, 2006.

[55] Michael Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *Int. J. Parallel Program.*, 35(3):233–262, June 2007.

[56] Jian Guan, Su Yan, and Jian-Ming Jin. An openmp-cuda implementation of multilevel fast multipole algorithm for electromagnetic simulation on multi-gpu computing systems. *Antennas and Propagation, IEEE Transactions on*, 61(7):3607–3616, 2013.

[57] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001.

[58] R. Higgins and A. Lastovetsky. Managing the construction and use of functional performance models in a grid environment. In *The 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 25-29, 2009 2009.

[59] Sungpack Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 78–88, 2011.

[60] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[61] Wen-Mei Hwu, K. Keutzer, and T.G. Mattson. The concurrency challenge. *Design Test of Computers, IEEE*, 25(4):312–320, 2008.

[62] A. Ilic and L. Sousa. On realistic divisible load scheduling in highly heterogeneous distributed systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 426–433, 2012.

[63] Intel Corporation. Intel Math Kernel Library. `https://software.intel.com/en-us/intel-mkl`.

[64] R I Issa. Solution of the implicitly discretised fluid flow equations by operator-splitting. *J. Comput. Phys.*, 62(1):40–65, January 1986.

[65] Honghoon Jang, Anjin Park, and Keechul Jung. Neural network implementation using cuda and openmp. In *Digital Image Computing: Techniques and Applications (DICTA), 2008*, pages 155–161, 2008.

[66] P. J. Joseph, K. Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 161–170, 2006.

[67] Maher Kaddoura, Sanjay Ranka, and Albert Wang. Array decompositions for nonuniform computational environments. *J. Parallel Distrib. Comput.*, 36(2):91–105, August 1996.

[68] Bo Kågström, Per Ling, and Charles van Loan. Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, September 1998.

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

[70] Alexey Kalinov and Alexey Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In *HPCN Europe 1999, LNCS 1593*, pages 191–200. Springer Verlag, 1999.

[71] George Karypis and Kirk Schloegel. ParMetis: Parallel graph partitioning and sparse matrix ordering library. `http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/manual.pdf`, 2013.

[72] Khronos Group. The OpenCL Specification. `http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf`, November 2013.

[73] V.V. Kindratenko, J.J. Enos, Guochun Shi, M.T. Showerman, G.W. Arnold, J.E. Stone, J.C. Phillips, and Wen-Mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, 2009.

[74] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[75] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning gemms for fermi. 2011.

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

[77] A. Lastovetsky and J. Dongarra. *High Performance Heterogeneous Computing*. Wiley, 2009.

[78] A. Lastovetsky and R. Reddy. Data partitioning with a realistic performance model of networks of heterogeneous computers. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 104–, 2004.

[79] A. Lastovetsky and R. Reddy. Data partitioning for multiprocessors with memory heterogeneity and memory constraints. *Scientific Programming*, 13:93–112, 2005.

[80] A. Lastovetsky and R. Reddy. Data distribution for dense factorization on computers with memory heterogeneity. *Parallel Computing*, 33:757–779, 12/2007 2007.

[81] A. Lastovetsky and R. Reddy. Distributed data partitioning for heterogeneous processors based on partial estimation of their functional performance models. In *7th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2009)*, pages 91–101, Delft, Netherlands, 25/9/2009 2010. Lecture Notes in Computer Science, vol. 6043, Springer, Lecture Notes in Computer Science, vol. 6043, Springer.

[82] A. Lastovetsky and R. Reddy. Two-dimensional matrix partitioning for parallel computing on heterogeneous processors based on their functional performance models. In *7th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar 2009)*, pages 112–121, Delft, Netherlands, 25/9/2009 2010. Lecture Notes in Computer Science, vol. 6043, Springer, Lecture Notes in Computer Science, vol. 6043, Springer.

[83] A. Lastovetsky, R. Reddy, and R. Higgins. Building the functional performance model of a processor. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 23-27 2006 2006. ACM, ACM.

[84] Alexey Lastovetsky and Ravi Reddy. Data partitioning with a functional performance model of heterogeneous processors. *Int. J. High Perform. Comput. Appl.*, 21(1):76–90, February 2007.

[85] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[86] A. Legrand, H. Renard, Y. Robert, and F. Vivien. Mapping and load-balancing iterative computations. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):546–558, 2004.

[87] Michael D. Linderman, Jamison D. Collins, Hong Wang, et al. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43:287–296, 2008.

[88] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55, 2009.

[89] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, July 2003.

[90] JA Martínez, EM Garzón, A. Plaza, and I. García. Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE. *J. Supercomput.*, 58(2):151–159, 2011.

[91] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[92] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, September 2012.

[93] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, November 2010.

[94] J. Nickolls and W.J. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[95] NVIDIA Company. CUBLAS manual. `http://docs.nvidia.com/cuda/cublas/#axzz3Ck6Wsm4t`, 2013.

[96] NVIDIA Company. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`, 2013.

[97] NVIDIA Company. NVIDIA Developer Zone. `https://developer.nvidia.com`, 2013.

[98] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *IPDPS 2008*, pages 1–10, 2008.

[99] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 4.0. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, May 2013.

[100] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[101] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[102] Phitchaya Mangpo Phothilimthana et al. Portable performance on heterogeneous architectures. ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM.

[103] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44(4):121–130, February 2009.

[104] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10, pages 217–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[105] Rolf Rabenseifner. Hybrid parallel programming on hpc platforms. In *Proceedings of the Fifth European Workshop on OpenMP*, EWOMP'03, pages 22 – 26, 2003.

[106] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436, 2009.

[107] Rolf Rabenseifner, Georg Hager, Gabriele Jost, and Rainer Keller. Hybrid mpi and openmp parallel programming. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 11–11, Berlin, Heidelberg, 2006. Springer-Verlag.

[108] Da Qi Ren, E. Bracken, S. Polstyanko, N. Lambert, R. Suda, and D.D. Giannacopulos. Power aware parallel 3-d finite element mesh refinement performance modeling and analysis with cuda/mpi on gpu and multi-core architecture. *Magnetics, IEEE Transactions on*, 48(2):335–338, 2012.

[109] Vladimir Rychkov, David Clarke, and Alexey Lastovetsky. Using multidimensional solvers for optimal data partitioning on dedicated heterogeneous HPC platforms. In *PaCT-2011*, volume 6873 of *LNCS*, pages 332–346. Springer, 2011.

[110] Gudula Rnger and Michael Schwind. Fast recursive matrix multiplication for multi-core architectures. *Procedia Computer Science*, 1(1):67 – 76, 2010. {ICCS} 2010.

[111] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[112] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[113] D.P. Scarpazza, O. Villa, and F. Petrini. Efficient breadth-first search on the cell/be processor. *Parallel and Distributed Systems, IEEE Transactions on*, 19(10):1381–1395, 2008.

[114] R. Sharma and P. Kanungo. Performance evaluation of mpi and hybrid mpi+openmp programming paradigms on multi-core processors cluster. In *Recent Trends in Information Systems (ReTIS), 2011 International Conference on*, pages 137–140, 2011.

[115] Fengguang Song, Stanimire Tomov, and Jack Dongarra. Enabling and scaling matrix computations on heterogeneous multi-core and multi-gpu systems. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 365–376, New York, NY, USA, 2012. ACM.

[116] Kyle Spafford, Jeremy S. Meredith, and Jeffrey S. Vetter. Quantifying NUMA and contention effects in multi-GPU systems. GPGPU-4, pages 11:1–11:7, New York, USA, 2011. ACM.

[117] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[118] The OpenFOAM Foundation. Numerical Schemes. `http://www.openfoam.org/docs/user/fvSchemes.php`, July 2014.

[119] The OpenFOAM Foundation. OpenFOAM User Guide. `http://www.openfoam.org/docs/user`, July 2014.

[120] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, 2010.

[121] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput.*, 36(5-6):232–240, June 2010.

[122] H. A. van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, March 1992.

[123] D.R. Vandenberg and Quentin F. Stout. Automatic hybrid openmp + mpi program generation for dynamic programming problems. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 178–186, 2011.

[124] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[125] C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comput. Syst.*, 17(5):601–623, 2001.

[126] John Robert Wernsing and Greg Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *LCTES 2010*, pages 115–124. ACM, 2010.

[127] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[128] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[129] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 19–28, 2010.

[130] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications*, 182(1):266 – 269, 2011. Computer Physics Communications Special Edition for Conference on Computational Physics Kaohsiung, Taiwan, Dec 15-19, 2009.

[131] Mohammed Javeed Zaki, Wei Li, and Michal Cierniak. Performance impact of processor and memory heterogeneity in a network of machines. In *In 4th Heterogeneous Computing Wkshp, also TR574*, 1995.

[132] Yao Zhang and J.D. Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393, 2011.

[133] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore platforms. In *2011 IEEE International Conference on Cluster Computing (Cluster 2011)*, pages 580–584, Austin, Texas, USA, Sept 26-30 2011. IEEE Computer Society, IEEE Computer Society.

[134] Z. Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications. In *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*, pages 191–199, Beijing, China, 24-28 September 2012.