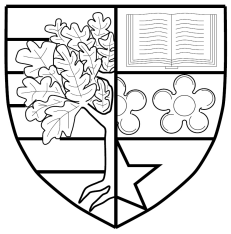
TOWARD OPTIMISED SKELETONS FOR HETEROGENEOUS PARALLEL ARCHITECTURE WITH PERFORMANCE COST MODEL

By

Khari A. Armih



Submitted for the Degree of

Doctor of Philosophy

at Heriot-Watt University

on Completion of Research in the

School of Mathematical and Computer Sciences

July, 2013

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Khari A. Armih (Candidate)

Greg Michaelson, Phil Trinder (Supervisors)

(Date)

Abstract

High performance architectures are increasingly heterogeneous with shared and distributed memory components, and accelerators like GPUs. Programming such architectures is complicated and performance portability is a major issue as the architectures evolve. This thesis explores the potential for algorithmic skeletons integrating a dynamically parametrised static cost model, to deliver portable performance for mostly regular data parallel programs on heterogeneous architectures.

The first contribution of this thesis is to address the challenges of programming heterogeneous architectures by providing two skeleton-based programming libraries: i.e. *HWSkel* for heterogeneous multicore clusters and *GPU-HWSkel* that enables GPUs to be exploited as general purpose multi-processor devices. Both libraries provide heterogeneous data parallel algorithmic skeletons including *hMap, hMapAll, hReduce, hMapReduce, and hMapReduceAll.*

The second contribution is the development of cost models for workload distribution. First, we construct an architectural cost model (CM1) to optimise overall processing time for *HWSkel* heterogeneous skeletons on a heterogeneous system composed of networks of arbitrary numbers of nodes, each with an arbitrary number of cores sharing arbitrary amounts of memory. The cost model characterises the components of the architecture by the number of cores, clock speed, and crucially the size of the L2 cache. Second, we extend the *HWSkel* cost model (CM1) to account for GPU performance. The extended cost model (CM2) is used in the *GPU-HWSkel* library to automatically find a good distribution for both a single heterogeneous multicore/GPU node, and clusters of heterogeneous multicore/GPU nodes. Experiments are carried out on three heterogeneous multicore clusters, four heterogeneous multicore/GPU clusters, and three single heterogeneous multicore/GPU nodes. The results of experimental evaluations for four data parallel benchmarks, i.e. sumEuler, Image matching, Fibonacci, and Matrix Multiplication, show that our combined heterogeneous skeletons and cost models can make good use of resources in heterogeneous systems. Moreover using cores together with a GPU in the same host can deliver good performance either on a single node or on multiple node architectures.

Acknowledgements

My praises to God for giving me the good health, the strength of determination and support to finish my work successfully.

I would like to thank my supervisor Professor Greg Michaelson, not only for his suggestions, but also for his friendship and his encouragement. His faith in me has been essential to overcome the hardest moments.

A special thank goes to my second supervisor Professor Phil Trinder for his helpful remarks regarding every aspect of my work.

Many thanks to the Libyan higher education sector for offering me this scholarship.

I am very grateful to the people I have been in contact with in Edinburgh and my country. These include my office mates and my close friend Gamal Alusta.

And last but not least, to my parents, my wife and my son for their affectionate support, encouragement and understating over the years.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

In recent decades a large variety of parallel system architectures have been introduced to the market. With the advent of multicore systems, parallelism has become mainstream. In terms of memory architectures, a cluster of multicore nodes can be classified as a combination of distributed and shared-memory parallel architectures. Such a parallel architecture is intended to increase performance by providing two levels of parallelism, one at the top level between the nodes and the second within each node in the cluster. However, in spite of the undoubted ability to increase the processing power of multicore clusters by upgrading their nodes or adding new nodes, this increase in processing power often results in heterogeneous environments due to the variety in the capabilities of the newly added or upgraded nodes.

Another promising parallel architecture is the Graphics Processing Unit (GPU) [2, 3]. An architecture that comprises manycores and GPUs is a highly efficient platform for both graphics and general-purpose parallel computing because it offers extensive resources such as high memory bandwidth and massive parallelism. This increases the degree of heterogeneity in clusters, as each node in the heterogeneous cluster may comprise a multicore and GPU elements. Such heterogeneous parallel architectures challenge the parallel language community to develop portable and efficient parallel programming models and languages.

Initially hybrid parallel programming models were introduced to exploit the strengths of such architectures. Much research [4, 5, 6, 7] has aimed to combine a distributed-memory programming model, such as message passing, between cluster nodes with a shared-memory programming model on each node. For heterogeneous multicore systems there are a number of General-Purpose Graphical Processing Unit (GPGPU) programming frameworks [8, 9, 10, 1] concerned with using only GPU, but not much work has been done to utilise both the Central Processing Unit (CPU) and GPU [11].

Algorithmic skeleton [12] reduces the complexity of programming hybrid and heterogeneous parallel architectures. Much work has been done in the area of skeletal programming [13, 14, 15] for distributed systems, but few skeleton frameworks [16, 17] have been proposed for heterogeneous parallel platforms.

The main goal of this thesis is to provide a high-level heterogeneous programming model that can hide from programmers the low-level details that are commonly encountered on heterogeneous parallel architectures. Thus programmers can concentrate on application-specific issues only. We propose to provide application developers with heterogeneous algorithmic skeletons that capture common parallel patterns of computation and communication, and use them to develop parallel applications on hybrid and heterogeneous architecture in a sequential manner. These heterogeneous skeletons are able to expose the underlying hardware and provide a suitable parallel programming model. The parallel implementation of these heterogeneous skeletons conceals almost all of the underlying hardware details and coordination activities i.e. communication, synchronisation, and load distribution. Moreover, these heterogeneous parallel skeletons provide even more flexibility by supporting various parallel architectures such as single- and multicore, multi-node, and integrated multicore/GPU parallel architectures.

Another contribution of this work is to propose and develop a performance cost model for workload distribution to achieve a good performance on a heterogeneous parallel architecture, by providing an efficient static load-balancing strategy. The proposed cost model is integrated in our heterogeneous skeletons to automatically guide the distribution of workload, which minimises the task of workload distribution for the skeleton programmer.

In this thesis, we present our methodology of designing and implementing

the *HWSkel* library for heterogeneous multicore clusters, and its extension *GPU-HWSkel* for heterogeneous multicore/GPU clusters to provide the necessary portability, using a hybrid programming model that employs Message Passing Interface (MPI) [18] for message passing, OpenMP[19] for multicore CPUs, and CUDA [1] for GPU programming. Our framework is written in C language due to the popularity of system level imperative languages in the parallel domain.

1.2 Research Contributions

The methodology proposed in this thesis is centred on providing a high-level skeleton-based programming framework, to simplify and improve the performance of data parallel programs on either heterogeneous *multicore* or *multicore/GPU* architectures.

1.2.1 Contribution Summary

The main contributions of this thesis can be summarised as follows:

- We provide surveys of parallel computing. First we survey parallel architectures and their programming models. Next we survey skeletal approaches, and GPU skeleton libraries and languages. Finally we survey performances cost models of parallel computing.

- We have designed a skeleton-based library called *HWSkel* for heterogeneous parallel hardware, in particular, for heterogeneous multicore clusters [20].

The library is implemented using a hybrid MPI/OpenMP programming model, where MPI is used as message passing between cluster nodes, and OpenMP is a shared memory programming model on each node. The *HWSkel* library provides a set of heterogeneous skeletons for data parallel computations *i.e. hMap, hMapAll, hReduce, hMapReduce, hMapReduceAll*.

- We develop a new architectural cost model (CM1) for load balance on heterogeneous multicore architectures [20]. The cost model characterises components of the architecture by the number of cores, clock speed, and crucially the size of the L2 cache. We demonstrate that the cost model can be exploited by skeletons to improve load balancing on heterogeneous multicore architectures. The heterogeneous skeleton model facilitates performance portability, using the architectural cost model to automatically balance load across heterogeneous components of the heterogeneous multicore cluster.

- We extend the *HWSkel* library with a CUDA-based skeleton library called *GPU-HWSkel*, which enable GPUs to be exploited as general purpose multiprocessor devices in heterogeneous multicore/GPU architectures. *GPU-HWSkel* uses a heterogeneous parallel programming model that employs MPI and OpenMP for CPU programming, and CUDA for GPU programming. The *GPU-HWSkel* library implements the same set of heterogeneous skeletons provided by the *HWSkel* libray to program parallel heterogeneous

multicore/GPU systems including both single- and multicore CPU, and GPU architectures.

- We construct a new cost model (CM2) based on the CM1 cost model to account for GPU performance, and we integrate this cost model into our heterogeneous skeleton to implicitly predict the costs of parallel applications. The new cost model is used to automatically find a good distribution for both a single heterogeneous multicore/GPU node, and clusters of heterogeneous multicore/GPU nodes. It is viewed as two-phase: the Single-Node phase guides workload distribution across CPU core and GPU using the performance ratio between the CPU and GPU in a multicore/GPU computing node; and the Multi-Node phase balances the distribution of workload among the nodes of a heterogeneous multicore/GPU cluster.

The following two sections outline the thesis contributions in more detail.

1.2.2 Addressing the Challenges for Heterogeneous Hardware Programming

In a homogeneous parallel architecture, developing parallel applications is a very complex process, where developers are required to explicitly manage all parallel coordination activities including the distribution, synchronisation, and communications patterns. Therefore, heterogeneous parallel architectures introduce

even more parallel activities, which further increase the complexity of developing parallel applications. The complexity of programming heterogeneous parallel architectures encompasses three challenges: *i)* ***Programmability***, the programming effort required to write and modify parallel programs for heterogeneous systems; *ii)* ***Portability***, the requirement that parallel programs written for heterogeneous systems should be portable across all the hardware that forms the heterogeneous system; *iii)* ***Performance***, finding the optimal load distribution ratio across the heterogeneous processing elements to improve the performance of parallel programs on the heterogeneous parallel platform.

In this section, we discuss the programmability and portability challenges, while the performance challenge is discussed in section 1.2.3.

Programmability. To exploit the strengths and improve the performance of a heterogeneous parallel system, a hybrid programming model is needed to provide different parallel programming models for different parallel architectures that might be part of the heterogeneous parallel system.

A hybrid programming model is a combination of two or more different programming models. For example, hybrid parallel programs for a heterogeneous multicore cluster require at least two programming models, the message passing model to communicate between the nodes, and the shared-memory programming model for multicore processing. A common example of this hybrid approach is the combination of MPI [21] as the message

passing model and OpenMP [19] for the shared-memory model. Furthermore, in a heterogeneous multicore/GPU cluster, we need another parallel programming model for GPU programming besides the distributed- and shared-memory programming models.

Despite these capabilities of a hybrid parallel programming model to improve the performance of software applications run on different heterogeneous parallel architectures, hybrid programming adds more complexity to parallel programming development. Usually developing one level of parallelism requires more effort than developing a sequential application as the applications developer is asked to explicitly handle all parallel activities such as data partitioning, communications, and synchronisation. For instance developing a parallel application using the MPI library requires considerable restructuring of the sequential program. So adding another level of parallelism to the same application will increase the level of complexity of developing such application due the management of the interaction between these two levels. In addition, using two or more different programming models in the same application require application developers to spend much time on extending their knowledge of writing and developing parallel applications.

Portability. Since programming a heterogeneous parallel architecture requires a hybrid programming model to provide various parallel implementations

for each parallel hardware unit in the system, hybrid programs which use such a hybrid model should be executable on each parallel hardware unit in the system independently.

This requires writing portable parallel program using a hybrid programming model that is smart enough to expose the underlying hardware and allows for a suitable parallel implementations in a transparent way. Writing such portable parallel programs which can be executed on a wide range of parallel systems (*i.e.* distributed- or shared-memory architecture) results in increasing of the degree of the complexity of using hybrid programming models.

The key idea of this work is to move the responsibility for dealing with the above challenges away from application developers by providing them with a high-level machine independent approach that is able to implicitly manage these challenges to simplify programming heterogeneous parallel architectures.

One promising approach to achieve our goal is to use high-level parallel heterogeneous skeletons that abstract away all parallel activities to reduce the complexity encountered in developing parallel applications. Skeletons are high-level abstractions that support widely used parallel programming patterns, where the control and the communications of parallel patterns are encapsulated in these abstractions. So skeletons are intended to simplify parallel programming by concealing all details required in parallel activities from applications developers and

allowing the developers to concentrate on high-level aspects of the parallel algorithm.

Due to the complexity of a hybrid programming model, skeletal programming can have a considerable impact by hiding the interactions between all the different individual parallel programming models. Moreover, parallel applications developers no longer need to learn any other parallel languages, and can write parallel programs just as they write sequential programs.

According to the above considerations, we have developed a new skeletons library named *HWSkel* that supports data parallel skeletons such as *hMap, hReduce, hMapReduce, and hMapReduceAll* on heterogeneous multicore clusters. The *HWSkel* framework is provided as a library of C functions which achieves parallelisation using MPI [21] and OpenMP [19]. *HWSkel* has been designed to implement a hybrid programming model by combining MPI as the distributed programming model with OpenMP for the shared programming model. Moreover, since our heterogeneous skeletons need to be invoked within an MPI initialisation, the *HWSkel* library provides wrapper functions (e.g *InitHWSkel*() and *TerminateHWSkel*()) for some MPI routines to keep the user away from using unfamiliar library functions within the skeletal programs.

An extension of the *HWSkel* library called *GPU-HWSkel* is presented in the

thesis to provide the developers with heterogeneous skeletons for parallel programming on more heterogeneous architectures, in particular, heterogeneous multicore/GPU clusters. The *GPU-HWSkel* library is implemented by a hybrid programming model comprising different programming models, including MPI and OpenMP for distributed- and shared-memory models, and uses the CUDA programming model to make a GPGPU accessible on NVIDIA GPUs. The library provides a portable parallel programming environment for a wide range of parallel architectures. *GPU-HWSkel* implements the same set of data-parallel skeletons that are provided by the base library. Our framework provides programmers with simple user functions, using macros to create CUDA kernel code for a GPU as well as C-like functions. These user functions can be passed as arguments to the skeletons.

1.2.3 Developing Performance Cost Models for Heterogeneous Skeletons

In general, parallel computational cost models are used in designing and optimising parallel algorithms and applications. They help the applications developers in understanding all important aspects of the underlying architecture without knowing unnecessary details. However, these computational models can play an important role in predicting the performance of a given parallel program on a

given parallel machine. In this section, we will discuss the performance challenges that are introduced by programming heterogeneous parallel architectures, proposing a performance cost model as a solution to overcome this challenge.

The level of complexity of designing and implementing efficient parallel algorithms for heterogeneous parallel platforms (*i.e.* a heterogeneous multicore cluster) is related to the degree of system heterogeneity. The diversity of the underlying hardware in heterogeneous systems has a remarkable impact on the performance of parallel algorithms. Thus, the performance of parallel algorithms depends on the ways of exploiting the distinct architectural characteristics of each target architecture, which is difficult on heterogeneous architectures, but manageable on homogeneous architectures.

So the challenge is how to design parallel algorithms that take into consideration this diversity of the underlying hardware to obtain a good performance of heterogeneous parallel systems. One way to overcome this challenge is to use a parallel computational cost model to guide the design of parallel algorithms and predict their performances on heterogeneous architectures. To predict the performance of a target architecture, performance cost models need to characterise the target architecture, by using performance parameters.

Our goal is, therefore, to provide a performance cost model to guide our heterogeneous skeletons to obtain a good performance on the target heterogeneous parallel systems. At the same time, we want to integrate this cost model into our heterogeneous skeleton to implicitly predict the costs of parallel applications. In

other words, we seek to provide the programmer with high-level machine independent and simple heterogeneous skeletons which allow for performance portability by using a performance cost model that provides cost estimations on a broad range of heterogeneous architectures.

We propose two performance cost models for two different heterogeneous parallel systems:

Heterogeneous multicore Cluster: We propose a static architectural performance cost model to guide our heterogeneous skeletons for workload distribution on a heterogeneous system composed of networks of arbitrary numbers of nodes, each with an arbitrary number of cores sharing arbitrary amounts of memory. The cost model supports performance portability by providing cost estimation on a broad range of heterogeneous multicore platforms. The proposed performance cost model is used to transparently and statically determine the data-chunk size according to the number of cores, clock speed and crucially the L2 cache size for each node over the heterogeneous cluster.

Heterogeneous multicore/GPU Cluster: We present another performance cost model that is based on the above-mentioned performance cost model to account for the GPU as an independent processing element and automatically find optimal distributions in heterogeneous multicore/GPU systems.

The model is viewed as a two-phase cost model since the underlying target hardware consists of two levels of heterogeneous hardware architectures. *i)* ***Single-Node Phase***, to guide the workload distribution across the CPU cores and GPU device inside each node in the *integrated* multicore/GPU system; *ii)* ***Multi-Node Phase***, to balance the workload amongst the nodes in the cluster.

1.3 Roadmap of The Thesis

This work is divided into two main parts. The first part concerns the proposed methodology for the design and implementation of our skeleton-based framework (Chapters 3, 5), and the second part deals with providing an efficient cost models to improve the performance of our heterogeneous skeletons on heterogeneous parallel architectures(Chapters 4, 6). The thesis is structured as follows:

Chapter 2 gives a survey of parallel computing. We first describe the parallel hardware architectures, giving an overview of the available parallel architectures in Section 2.2. We then give an overview of existing common parallel programming models, describing the use of these programming models for available parallel architectures in Section 2.3. Finally, we present the concept of using skeleton programming in parallel and distributed computing, and give an in-depth description of existing skeleton-based frameworks in Section 2.4.

Chapter 3 presents a C skeleton-based library (*HWSkel*). We describe and discuss the parallel implementation of the heterogeneous skeleton in the *HWSkel* library in Section 3.1. We also present a description for each skeleton and its interface in the current *HWSkel* prototype in Section 3.2.

Chapter 4 describes our proposed architecture-aware cost model for heterogeneous multicore clusters. The chapter starts by providing a survey of cost models in parallel computing in Section 4.1. We discuss some resource metrics that are visible in all parallel computational models in Section 4.2 Section 4.3 presents the design ethos of CM1 cost model. We then discuss our approach to developing a cost model to optimise overall processing time in Section 4.4. We discuss how to integrate the cost model into our heterogeneous skeletons to improve their parallel performance in Section 4.5. We finish the chapter by discussing experimental results to evaluate the effect of the cost model on the performance of our heterogeneous skeletons in Section 4.6.

Chapter 5 introduces our extended *GPU-HWSkel* library. The chapter starts by introducing GPU programming, giving an overview of the available parallel models for GPGPU computing in Section 2.3.3. First we survey GPU skeletal approaches, and GPU skeleton libraries and languages in Section 2.4.2.4. Next we describe the design and implementation of the *GPU-HWSkel* library in Section 5.1.

Chapter 6 presents the extended cost model that is integrated into the *GPU-HWSkel* library. We survey work on cost models for multicore/GPU systems in Section 6.1. We discuss our approach in Section 6.2. Next we present the extension of our multi-processor/multicore model to multicore/GPU in Section 6.3. The chapter is concluded by investigating the ability of our cost model to improve the performance of our heterogeneous skeleton in different heterogeneous environments in Section 6.4.

Chapter 7 concludes giving a summary of the work and outlining directions for future work. It also discuss the limitations of this work.

1.4 Publications

The work reported in this thesis led to the following publications:

- K. Armih, G. Michaelson and P. Trinder, Cache Size in a Cost Model for Heterogeneous Skeletons, Proceedings of HLPP 2011 : 5th ACM SIGPLAN Workshop on High-Level Parallel Programming and Applications, Tokyo, September 2011.

- K. Armih, G. Michaelson and P. Trinder, Heterogeneous Skeletons for CPU/GPU Systems with Cost Model, *Submitted* to HLPP 2013: International Symposium on High-level Parallel Programming and Applications, Paris, July 2013.

Chapter 2

Background

This chapter provides a survey of parallel computing. The focus of this chapter is on parallel architectures and their programming models. The chapter opens with a description of parallel hardware architectures and subsequent sections give an overview of common parallel programming models, describing the use of these programming models on the available parallel architectures. Next the chapter provides a survey of algorithmic skeleton frameworks, giving an in-depth description of the current skeleton frameworks. Finally, the chapter closes with discussion of open problems that summaries the contributions of this thesis.

2.1 Parallel Computing

Simply put, parallel computing is solving big computational problems using multiple processing elements simultaneously rather than using a single processing

unit.

The main target of parallel computing is to solve large problems that do not fit in one CPU's memory space, and to achieve good performance on parallel systems by reducing the execution time. To accomplish these targets, a computational problem is decomposed into independent sub-problems that can be executed simultaneously using parallel systems.

Parallel systems can be a single computer with multiple processors, multiple computers connected through a local network, or a multicore system that has two or more cores in one single package.

2.2 Short Survey of Parallel Architectures

Parallel computing systems comprise multiple processing elements connected to each other through an interconnection network plus the software needed to make the processing elements work together [22].

The most common way to classify parallel system architectures is Flynn's taxonomy [23]. His classification is based upon the number of instruction streams and data streams available in the architecture.

However, we here classify the parallel system architectures according to the way each processing element in the system accesses the memory.

2.2.1 Distributed Memory Architectures

Each processing element in a distributed memory system has its own address space and communicates with others by message passing.

- **Multiprocessor Architectures**

  A multiprocessor is a parallel computing system that consists of multiple processing elements connected to each other through an interconnection network. These systems support either shared memory or distributed memory. Figure 1 and Figure 2 show the structure of both shared memory multiprocessors and distributed memory multiprocessors [24].

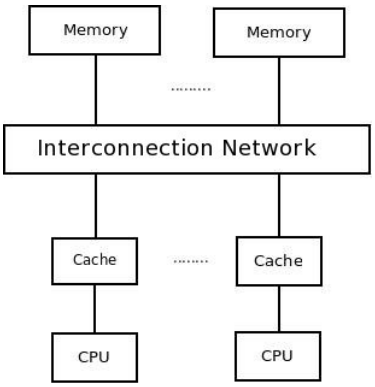


Figure 1: Structure of a Shared Memory Multiprocessor

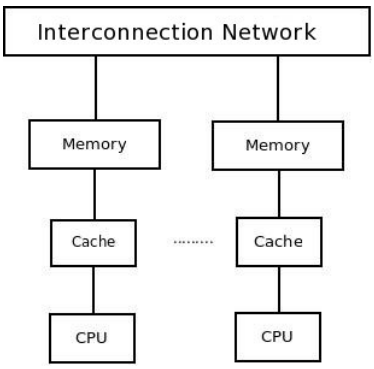


Figure 2: Structure of a Distributed Memory Multiprocessor

- **Clusters**

  A cluster is a distributed memory system that consists of several standalone computers connected to each other by a local area communication network. A Beowulf cluster [25, 26] is a widely used multi-computer architecture, composed of off-the-shelf compute nodes connected by fast Ethernet or some

other network. A common Beowulf system is built out of standard PCs that run free software, such as the Linux operating system, Parallel Virtual Machine (PVM) [27] and MPI [21]. Therefore, such systems can provide cost-effective way to gain fast and reliable services.

- **Grid**

  A grid [28] is a distributed memory system composed of many supercomputing sites connected by the Internet into a single system to provides considerable computational resources (such as I/O capacity, memory, or computing power) to a range of high performance applications. Applications in such an integration of computers benefit from a uniform and simplified access to the available resources. A grid can be viewed as a high-performance parallel system since it provides multiple levels of parallelism.

2.2.2 Shared Memory Architectures

In shared memory architectures, all processing elements share a single address space and communicate with each other by writing and reading shared memory locations.

- **Symmetric Multiprocessors**

  A Shared Memory Multiprocessor (SMP)[22] is a shared memory system which consists of multiple processors connected to each other through an interconnection network, where all processors share the main memory and

have equal access time to all memory locations. In a SMP system each processor has its own cache memory as shown in Figure 1. In terms of the programming model, SMP systems are the easiest parallel systems because there is no need to explicitly distribute the data amongst the processors.

- **Multi-Core Architectures**

  A multicore architecture (or Chip Multiprocessor(CMP)) is a special kind of shared memory multiprocessor system which consists of two or more independent computational cores. These cores are embedded into a single chip processor [29]. Each core has a small private L1 cache and shares L2 cache and global memory with other cores. The common model of multicore is dual-core (e.g. Intel's Core 2 Duo [30], AMD Athlon X2 6400+ dual-core [31]) which contains two cores in a single die. The other models such as quad-core (e.g. Intel Xeon Processor E5410 [32]), eight-core (e.g. Intel 8-core Nehalem-EX [33]) models come as forms of multiple single-die of dual-core models, and the 48-core Intel SCC processor [34] that consists of 24 tiles of dual-core each.

  In contrast with symmetric multiprocessor(SMP), multicore is more efficient due to reduced memory bandwidth bottlenecks and communication complexity. Since the cores in a multicore systems are closely tied together, the system can take advantage of fast on-chip communication and higher bandwidth among the cores.

2.2.3 GPU Architectures

GPU is a special processor used to manipulate computer graphics. Recently, GPUs have shown great computational capabilities over the CPU [35].

In contrast, CPUs consist of few computational cores that are designed to support a wide range of applications. This limit of the number of cores restricts the number of data elements that can be processed concurrently. On the other hand, GPUs consist of hundreds of small cores which yields massive parallel processing capabilities; this increases the number of data elements that can be processed simultaneously. Due to the natural parallel architecture of GPUs, GPGPU have became the most efficient, cost-effective platform in parallel computing. Architecturally, a GPU is used as a co-processor to accelerate the CPU for GPGPU by executing the serial part of the program on the CPU and the parallel part on the GPU [36, 35].

In this thesis we focus on NVIDIA architectures. NVIDIA GPUs [37] have a number of multiprocessors, which can be executed in parallel. There are a variety of NVIDIA GPUs of different architectures. For example, the NVIDIA Tesla GPU [38] consist of a number of multiprocessors constructed from 8 scalar processors, where the multiprocessor in the NVIDIA Fermi GPU has two groups of 16 scalar processors. Figure 3 shows a simple view of the NVIDIA GPUs architecture model. NVIDIA GPUs are based on an array of Streaming Multiprocessors (SMs). Each multiprocessor consist of a number of Scalar Processor(SP) cores. A small

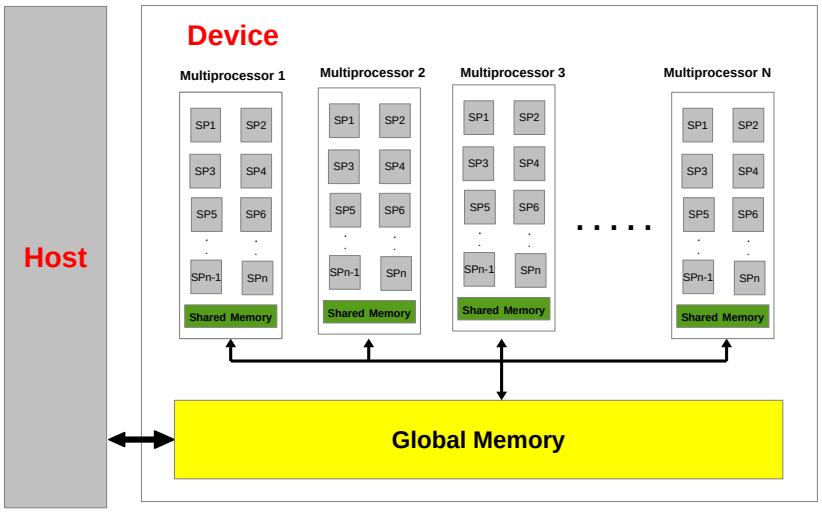


Figure 3: Simple Model of a GPU Architecture

local memory (e.g. 16KB) (referred to as shared memory) is integrated into each multiprocessor to be shared by all SP cores. All streaming multiprocessors are connected to large global memory(DRAM). The CPU has access(read/write) to various types of memories on the GPU namely global, constant and texture memory.

Furthermore, like the classical Single Instruction Multiple Data (SIMD) processors, all SP cores in the same multiprocessor execute in SIMT (Single Instruction, Multiple Thread) fashion, where each core can execute a sequential thread.

2.2.4 Heterogeneous Parallel Architectures

Heterogeneous or hybrid systems are high-speed parallel computing systems that consist of several nodes with separate address spaces(multicore cluster), where each node contains multiple processing elements that shared the same memory address.

Heterogeneous systems can take different architectural forms. They can be a small cluster of multicore (e.g dual-core) PCs or a large cluster of SMP nodes. A good example of cluster of SMP is the Earth Simulator system, which consists of 640 processor nodes (SMP nodes) that are connected through a high-speed network, where each node comprises 8 vector processors [39, 40].

Furthermore, with the advent of the graphics processing unit (GPU), general purpose computing on GPU (GPGPU)[2] has become popular in the parallel community and more complex heterogeneous systems *i.e.* multicore/GPU architecture are introduced. Most personal computers consist of a GPU that is connected to multicores via a PCI Express connection, which offers multi-level of hardware parallelism.

Currently, the High-Performance Computing (HPC) community tries to increase the number of processing elements to provide a high performance capability by using heterogeneous computing system architectures that are constructed from both CPUs and GPUs (e.g. heterogeneous *integrated* multicore/GPU cluster).

2.3 Parallel Programming Models

Parallel programming models present an abstract view of the underlying hardware. There are several different parallel programming models for different parallel architectures. These models can be implemented either in multiprocessor architectures as distributed memory programming models or in multicore architectures as shared memory programming models. Therefore the choice of a suitable parallel programming model depends on the underlying hardware of the system. Moreover, various parallel programming APIs for both distributed and shared memory parallel programming models have been introduced, to ease building efficient parallel applications [19, 18].

2.3.1 Distributed Memory Programming Models

The main distributed memory programming model is the message passing programming model, which is implemented on multiprocessor architectures where the underlying hardware is composed of a collection of processing elements [41]. The message passing model is based on a set of processes that use local memory, and communicate with each other by explicitly sending and receiving messages. The shortcoming of the message passing model is that the programmer is required to explicitly implement some tasks of parallelisation such as inter-process communication and synchronisation, and data partitioning and distribution.

However, the message passing model is widely used in parallel programming

due to several advantages. First, it supports Multiple Instruction Multiple Data (MIMD) architectures such as clusters, and also it can be implemented on SMP systems by using shared variables as message buffers. Second, the programmer has full control of data locality [41, 42]. The most common realisations of the message passing model are MPI [18], PVM [27], and Unified Parallel C(UPC) [43].

The Distributed Shared Memory (DSM) model [44] is another example of a distributed memory programming model. The basic idea of a DSM system is to implement a shared memory programming model on distributed memory systems. Since shared memory programming provides a straightforward way of programming and portability, DSM is intended to take full advantage of the shared memory programming model capabilities to achieve good scalability and performance on distributed memory architectures.

There are several DSM implementations[44, 45] including IVY, Midway, Clouds, and Munin.

- **MPI**

  MPI [21] is an API (Application Programming Interface) for the distributed memory message passing programming model. It provides the user interface and functionality for message passing capabilities. MPI is a library, not a language, that provides language bindings for C, C++, and Fortran. The MPI standard includes subroutines that provide a message passing

environment for distributed memory architectures, and even shared memory systems. It delivers good portability, since it provides high flexibility of implementation on a broad variety of architecture. Moreover, the MPI implementation was designed to support heterogeneity, which means that MPI has the ability to function on heterogeneous architectures.

Unfortunately, MPI has a number of disadvantages [21, 18, 46]. The main disadvantages are related to low-level communication, where sending and receiving data between processors can often create a large overhead, which has to be minimised. For example, the granularity often has to be large, since the fine grain granularity can create a large quantity of communications. And also dynamic load balancing is often difficult. Furthermore, creating and developing parallel programming based on MPI requires much effort, since the developer must take the responsibility of creating the code that will be executed by each processor.

2.3.2 Shared Memory Programming Models

Shared memory, or multi-threaded programming models are supported on shared memory multiprocessor architectures described in Section 2.2.2, where the underlying hardware has multiple processing elements, which access the same shared memory. A shared memory program is executed by independent threads that access the same shared address space. Shared memory models employ shared

variables for communication and synchronisation between threads. Quinn [42] says that a common approach to a shared memory model is fork/join parallelism, where the program starts with one thread to execute the sequential part of the algorithm and then creates a team of threads to execute the parallel part. At the end of the parallel part all threads are destroyed and the program returns to a single thread. One of the important capabilities of the shared memory model is the ability to support incremental parallelisation. There are several realisations of shared memory programming models such as POSIX Threads [47], OpenMP [19], GpH [48, 49], Intel TBB [50], and FastFlow [51].

- **OpenMP**

  OpenMP is a shared-memory application programming interface(API) that is suitable for implementation on shared memory multiprocessor architectures. It gives the programmer a simple and flexible interface to parallelise sequential applications. In order to take full advantage of shared memory programming model capabilities, OpenMP was designed to support fork/join parallelism. Moreover, OpenMP was created to support incremental parallelisation to achieve good performance [19].

  OpenMP consists of a set of compiler directives and runtime library functions to identify parallel regions in Fortran, C, or C++ programs. The directives were provided to control parallelism; therefore the programmer can tell the compiler which parts of existing code will be run in parallel

and how to distribute the tasks among the threads using these directives. OpenMP has become widely used in shared memory programming due to the simplicity of use. All the details of parallel programming are up to the compiler, and also it is well-structured for parallel programming.

- **Pthreads**

  Pthreads or Portable Operating System Interface for Unix (POSIX) Threads is an API for creating and manipulating threads. It was created to support shared memory programming models. The Pthreads standard is defined as a library of C programming language types, functions and constants to create, manipulate and manage threads [47]. For parallelising a sequential application using the Pthreads standard, the programmer is required to explicitly create, manipulate and manage threads. In contrast with OpenMP, using Pthreads is much more complicated, and the parallel code looks very different from the original sequential code.

2.3.3 GPU Programming

GPUs were designed as specialised processors to accelerate graphics processing. Recently, however, the architectures that comprise multicores and GPUs have become ubiquitous and cost effective platforms for both graphics and general-purpose parallel computing, as they offer extensive resources such as high memory bandwidth and massive parallelism [3]. Compared to a CPU, the performance

of a GPU comes from creating a large number of lightweight GPU threads with negligible overheads, where in the general purpose multicore the number of cores limits the number of data elements that can be processed simultaneously.

With the programmability available on the GPU, a new technique called GPGPU (General Purpose computation on GPU) has been developed [2, 3]. GPGPU allows the use of a GPU to perform computing tasks that are usually performed by a traditional CPU. Many parallel applications have achieved significant speedups with GPGPU implementations on a GPU over the CPU [52]. However, GPGPU architectures can deliver high performance only to data parallel programs that have no inter-thread communication, so programs that incur frequent synchronisation during execution time have serious performance penalties.

All GPU programs follows a (SIMD) programming style, where many elements are processed in parallel using the same program. The NVIDIA GPU model [3] is SIMT (Single Instruction Stream, Multiple Threads), where the GPU program consists of sequence of code execution units called kernels. Each kernel is executed simultaneously on all SMs, and finishes before the next kernel begins by using implicit barrier synchronisation between kernels. These kernels comprise a number of lightweight GPU threads that are grouped in independent blocks; each thread is executed by a single processor and cannot communicate with any other.

Today's GPUs enable non-graphics programmers to exploit the parallel computing capabilities of a GPU using data parallelism. Consequently, a number of data-parallel programming languages [1, 53, 8, 9, 10] have been proposed for using GPUs for GPGPU by providing all the means of a data parallel programming model.

In this chapter we focus on the C-like CUDA language [1] introduced by NVIDIA that supports a general purpose multi-threaded SIMD model for GPGPU programming. Other programming languages that support GPGPU programming include Sh[53], which was proposed as a high-level shading language and implemented as C++ library on both GPU and CPU.

Brook[8] is high-level programming environment that views the GPU as a stream co-processor. Brook is implemented in C by providing data parallel operations for GPGPU programming; these operations are organised as a collection of kernels. A kernel is a call to a parallel function that takes one or more streams as input and produce one or more streams. In Brook, programmers are required to divide the program into kernels and handle GPU resources.

Similar to Brook, Microsoft Research introduced an advanced system called Accelerator[9], which provides a high-level programming model of GPU programming, but instead of the programmers dividing the program into kernels, Accelerator automatically divides the computation into pixel shaders. Accelerator uses the C# library to provide high-level data parallel array operations; the array operation sequences is transparently compiled to GPU kernels.

The Khronos group developed an open low-level standard called Open Computing language (OpenCL) [54, 10] for GPGPU programming by providing a high-level programming framework that supports various computing devices, enabling the programmers to exploit the parallelism of heterogeneous systems. OpenCL offers a unified way to achieve a high degree of software portability. Therefore it can enable programmers to write code not only for GPUs, but for CPUs and other compute devices. OpenCL is based on C for writing computing kernels. The current specification supports both data and task-based parallel programming models.

Some companies also provide programming frameworks for their own GPUs such as ATI Stream[55] for AMD GPUs.

- **CUDA Programming Model**

  CUDA (Compute Unified Device Architecture) [1, 3] is a popular parallel programming model that support parallel programming on GPUs. In particular, CUDA simplifies the parallel applications development on NVIDIA GPUs by providing a data parallel programming model that views the GPU as a parallel computing co-processor that can execute a large number of threads simultaneously.

  Like OpenCL[56], CUDA provides a set of extensions and a runtime library for C, but in the latest version of CUDA, a big subset of C++ is included, which makes CUDA programming much easier than OpenCL programming.

CUDA parallel systems consist of two components, the host (i.e. CPU), and the GPU. The GPU is organised as a grid of thread blocks as shown in Figure 4. A thread block can be executed on only one multiprocessor

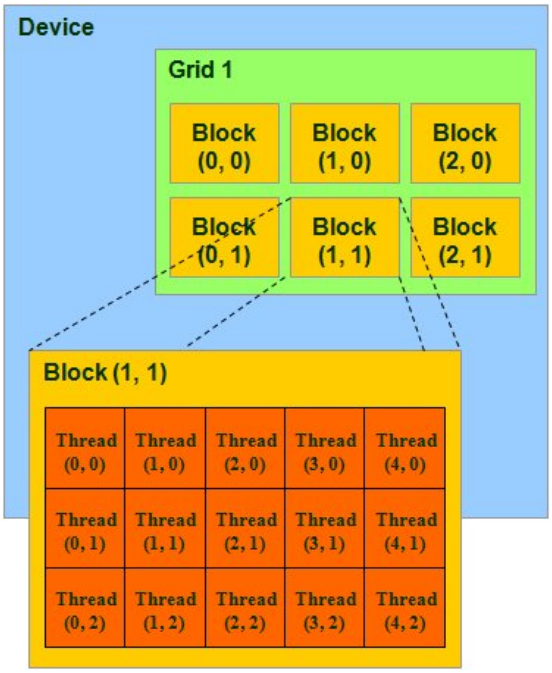


Figure 4: CUDA Architecture [1]

(SM), but one or more thread blocks can be executed simultaneously by a single SM in the CUDA-supported GPU device. A thread block is a set of threads that work together by sharing the data and they can synchronise with each other via barriers. The maximum number of threads inside each thread block is up to 1024 depending on the GPU computing capability.

A CUDA program is a single file of mixed source code for both CPU and

GPU implementations. The program is compiled by the NVIDIA C Compiler (NVCC) to separates the two codes. The *host* code represents the sequential phases (with little or no parallelism) that consist of one or more sequential threads, which are executed on the CPU. On the other hand, the parallel phases are executed on the GPU as *device* code that consists of a set of parallel kernel functions. Each kernel function executes a sequential program on a bunch of lightweight parallel threads that are organised in the form of a grid of thread blocks, where the number of threads inside each thread block and the number of thread blocks are specified within each kernel. In the CUDA model, the kernel function is executed in SIMD style. Since the CPU and GPU have separate memory address spaces, memory management such as transfer data between the CPU and the GPU have to be done explicitly using the CUDA-provided API functions.

The execution of a CUDA program starts on the CPU. Once the kernel function is invoked, the flow of the execution is moved to the *device* GPU, and then a bunch of lightweight parallel threads are generated and grouped as a grid while the data is transferred between the CPU and the GPU. After all threads of the running kernel complete their execution, the running thread grid is terminated and the flow of execution is moved back to the CPU. Since the CUDA program might have more than one kernel, the flow of program execution will keep switching between the CPU and the GPU depending on how many kernels are left in the running program.

2.3.4 Hybrid Programming Models

A hybrid programming model is a combination of different parallel programming models. This programming model can be used to take full advantage of the heterogeneous parallel architecture described in Section 2.2.4, where the underlying hardware consists of several nodes with separate address spaces (cluster), and each node contains a different number of multiple processing elements that share the same memory address space.

A common example of this combination is a hybrid programming model that uses shared- and distributed-memory parallel programming models[57]. This combination uses both a message passing model (MPI) and shared-memory model (OpenMP) or Pthreads; OpenMP is used to exploit the multicore per node while MPI is used to communicate between the nodes.

Another heterogeneous programming model is that based on CPU and GPU programming to provide support on heterogeneous CPU/GPU architectures. This heterogeneous programming model will be discussed in more detail in Chapter 5.

2.4 High Level Parallel Programming Approaches

Parallel architectures offer impressive computational capabilities compared to sequential architectures. Unfortunately, writing programs for those parallel architectures introduces much programming complexities to the programmers. Subsequently, the complexity of parallel programming causes programmers to focus

on the low-level parallel activities instead of concentrating on application-specific issues. The low-level parallel activities encompass all low-level details that are involved in the parallel application development process, where the nature of these activities differs from one parallel architecture to another. This includes activities such as: problem distribution, communication and synchronisation, data packing and unpacking, and load balancing.

Therefore, providing high-level parallel abstractions to enable programmers to ignore all these low-level implementation details during the development of any parallel programs is required to reduce the complexity of parallel programming [12, 58].

Design patterns are a valuable way to reduce the complexity of parallel programming [59]. Such high-level parallel abstractions are intended to capture common parallel patterns by hiding most of the low-level details from programmers, and can used as fundamental building blocks in parallel programs.

Several approaches have been introduced for raising the level of abstraction in parallel programming. These high-level abstraction-based approaches have been developed as:

- **New Languages:** A number of approaches such as llc [60] and P3L [61] have built a new language from scratch to raise the level of abstraction. This new language is based on an existing programming language as the host language. Parallel abstractions are provided through the language

syntax. The only drawback to this approach is that programmers need to learn a new language.

- **Parallel Compilers:** Providing a parallel compiler from existing language is another approach to high-level parallel programming such as Skil [62] and HDC [63]. A parallel compiler is used to compile the sequential program that contains high-level parallel constructs.

- **Library of Parallel Constructs:** A common high-level parallel programming approach such as Skel [64], Muesli [65], and Muskel [66] to extend an existing language with predefined parallel library constructs. From the viewpoint of the application programmer, this approach is much easier than other approaches.

2.4.1 Skeleton Programming

Structured parallel programming (or algorithmic skeletons) was first introduced by Cole in [12]. Conceptually, skeletons are designed to capture the common algorithmic parallel patterns of computation and communication, and are used to develop parallel applications. Therefore, the programmer can build parallel programs by using these parallel skeletons in a sequential manner, and also more complex programs by nesting the basic skeletons together.

Skeletons can also be treated as generic program components that implement

parallel patterns of computation; therefore they can be reused to develop different applications. In the parallel computing domain, the development of parallel programs is a hard and long process, therefore using a particular skeleton in developing several applications can offer the great advantage of investing more time in concentrating on application-specific issues.

According to the parallel patterns that are implemented by skeletons, parallel skeletons can be split into two classes: **i)** Data parallel skeletons which capture the patterns of data parallel computations, where structured data are partitioned among processing elements and computations are performed simultaneously on different parts of data structures. This includes *map, reduce, etc* [67]; **ii)** Task parallel skeletons that capture the parallelism from executing several unrelated tasks. This includes *pipe, farm, etc.* Skeletons in both classes can be integrated to provide more flexibility in developing complex applications [68].

Algorithmic skeletons can be provided to the user either as language constructs or as parallel library constructs. Four manifesto principles are presented in [64] and its extension in [69] to provide guidance to the future design of skeletal programming frameworks:

1. ***propagate the concept with minimal conceptual disruption***, skeletons have to be provided within existing programming languages without adding any new syntax.

2. ***integrate ad-hoc parallelism***, skeletons must be constructed in a way

that allow the users to integrate the skeletons to capture the parallel patterns that are not supported by the available skeletons.

3. ***accommodate diversity***, in constructing skeletons we have to ensure a balance between the need of abstraction for simplicity and the for realistic flexibility.

4. ***show the pay-back***, we should be able to show that skeletal programs can be easily executed on new architectures.

5. ***support code reuse***, that is skeletons can be used with no modification and minimum of effort.

6. ***handle heterogeneity***, skeletons should be implemented in a way that skeleton-based programs can be executed on heterogeneous architectures.

7. ***handle dynamicity***, skeletons should be able to handle dynamic situations, such as load balancing strategies.

Much work has been done in the area of skeletal programming under various names, and for different parallel architectures. Good general surveys of early research are given in [13, 14], and more recent detailed ones in [15, 70].

2.4.2 A Survey of Structured Parallel Programming Frameworks

Several implementations of structured parallel programming are available for the parallel community. They use different host languages and provide different sets of parallel constructs.

Since the organisation of parallelism in skeletal programming is up to the skeleton implementation, so algorithmic skeletons can be classified by their implementations on supported underlying hardware either as distributed- or shared memory architecture. This section surveys the major work done on structured parallel programming based on their implementations.

Note that all frameworks, and their constructs and skeletons names, are written as they have been by the original authors.

2.4.2.1 Distributed Computing Environment

Most of the high-level parallel programming approaches have been proposed for distributed computing architectures such as Clusters and Grids. These approaches have been developed either as new languages based on skeleton constructs or as parallel library constructs in well-known sequential languages such as C/C++ and Java.

Most of the approaches, if not all, provide support for both task-parallel and data-parallel skeletons.

P3L Language

P3L [61, 71] is structured high-level parallel programming language based on skeleton constructs. It uses the C language as host language along with a template-based compiler to implement P3L applications.

P3L provides the user with a set of parallel constructs that abstract the common forms of data and task parallelism. Skeletons in P3L include the sequential, farm and pipeline skeletons for task parallel implementations and, three skeletons for data parallelism namely map, reduce and comp[67].

Parallel implementation in P3L is based on implementation templates that are part of the P3L compiler, these templates have corresponding parallel constructs in P3L programs. Each construct in P3L has a number of templates that are implemented for different architectures. The P3L compiler uses these implementation templates to generate the final code.

A performance cost model is used by the compiler to guide the resource allocation process for each template to ensure optimisation for a given parallel architecture.

Skil

Skil[62] is an imperative language that uses algorithmic skeleton for parallel implementations. Some functional features such as higher-order functions, function currying, and polymorphic types which enable skeletons to be reused to solve related problems, are provided to integrate the skeletons into Skil. A subset of

the C language is employed as the basis of Skil to provide these features.

Skil provides a number of skeletons for data parallelism which work on the homogeneous data structure "distributed array". These skeletons include array-map, array-fold, array-gen-mult, array-permute-rows and other process parallel skeletons such as array-create, array-destroy, and array-copy skeletons. A new enhancement is presented in [72] to enable the algorithm skeletons to work on dynamic distributed data structures.

Skil comes with its own compilers. The function of this compiler is to process a Skil program with the provided skeletons by translating all high-order functions that implement parallel code into C target code.

HDC

HDC [63] is a functional language that uses a subset of the Haskell language as its host language. It use a higher-order functional programming style to implement skeleton programming. A compiler is provided for HDC.

HDC is designed for the parallelisation of the divide and conquer paradigm. Within HDC, skeletons for divide and conquer have different implementations, starting with general divide-and-conquer (dcA), fixed recursion depth (dcB), constant division degree (dcC), multiple block recursion (dcD), element-wise operations (dcE) and correspondent communication (dcF).

A HDC source program is processed by the compiler and translated into C code, where the code contains calls to MPI routines that handle communications

and memory management.

Muesli

Herbert Kuchen has provided algorithm skeletons[65] in the form of the C++ Muesli library that is implemented on the top of MPI. It takes advantage of polymorphism, higher order function and partial application concepts that exist in C++ to implement the skeletons.

This library offers data parallel and task parallel skeletons that can be nested [68] according to P3L's two tier[73] approach. Muesli provides parallel Skeletons for task parallel computations such as: Pipeline, Farm, DivideAndConquer, and BranchAndBound; where data parallelism is based on a distributed data structure for arrays, matrices, and sparse matrices to support data parallel skeletons such as: fold, map, and zip.

An optimisation technique is introduced in [74] for data parallel skeletons such as map and fold, where sequences of communication skeletons are combined to improve the performance of the skeleton implementation.

Lithium

Lithium [75] is a fully nestable algorithmic skeleton library based on Java. It exploits a macro data flow execution model to implement skeleton programs.

Skeletons in Lithium support both data and task parallelism, including a pipe skeleton, farm skeleton, iterative skeletons (Loop and While), data parallel

skeletons (Map and DivideConquer), and a conditional skeleton(If).

In the macro data flow model, the skeleton program is compiled into a data flow graph, where each node in the graph represents a sequential portion of code; therefore, each processing element can compute any one of these nodes.

An optimisation technique is presented in[76] to improve the performance of Lithium implementations in grid environments. This technique is integrated into the data flow model to ensure load-balancing, reduce the communication time, and hiding remote call latency by overlapping communications and computations.

eSkel

eSkel(**E**dinburgh **S**keleton **L**ibrary) is a library of C functions running on the top of MPI. The initial prototype was introduced in [64] with four principles of skeleton design guidance. In the second version of eSkel two fundamental concepts of skeleton definition were introduced[77, 78]: (*i*) *nesting-mode* where skeleton nesting can be transient or persistent; (*ii*) *interaction-mode* where the skeletons can interact implicitly or explicitly.

eSkel adopts the Single Program Multiple Data (SPMD) distributed model from MPI for its implementations, and all its skeleton operations are working within the MPI environment.

The eSkel library offers data parallel and task parallel skeletons for distributed environments such as clusters and grids. This includes skeletons such as Pipeline, Deal, Farm, HaloSwap and Butterfly. Besides, eSkel provides a data model(eDM)

for communicated data within the MPI environment.

Muskel

Muskel[66] is a full Java library for skeletal parallel programming. It provides nestable skeletons for distributed-memory environments such as clusters and grids.

Muskel provides a subset of parallel skeletons for both data and task parallel computations including farm, pipeline, and map skeletons.

Muskel is different to other skeleton implementation libraries apart from Lithium [75], where it takes advantage of macro data-flow technology to implement the algorithm skeletons instead of relying on the templates. This technology gives the users good flexibility in using the predefined skeletons or defining new skeletons for parallel computation patterns that are not covered by the existing skeletons.

In the current Muskel version[79], annotations and AOP (Aspect-Oriented Programming) techniques are introduced to abstract non-functional features of parallel programs such as security issues, source-to-source program optimisation rules, and data management (e.g. load-balancing).

SkeTo

SkeTo [80] is a C++ library that supports distributed parallel computations in a sequential way. This library uses MPI to achieve parallel distributed computations on distributed memory architectures.

SkeTo employs the theory of constructive algorithmic [81] to define its algorithm skeletons, where the computation structure is defined in terms of its data structure. Therefore, it provides constructive skeletons for data structures such as lists, trees, and matrices instead of providing parallel computation skeletons. Each of these skeletons is composed of three basic parallel skeletons namely *map*, *reduce* and *scan*. SkeTo uses a meta mechanism to define new skeletons, where a new skeleton can be constructed from the basic skeletons or the skeletons that have been built upon the basic skeletons.

SkeTo implemented a fusion transformation technique [82] with OpenC++ to improve skeleton performance. This technique is used to optimise the combinations of skeletons by eliminating any intermediate data structure that is generated between skeleton calls and reducing the overhead of the skeleton calls. In the new version of SkeTo[83] the fusion transformation technique is implemented in C++ using an expression template technique.

Calcium

Calcium [84] is a Java library for parallel skeletons, which uses the ProActive environment [85] to achieve parallelism on parallel distributed architectures. ProActive is Java middleware which is used for parallel distributed programming.

Calcium provides basic parallel skeletons for both task and data parallelism. Skeletons in Calcium can be nested and applied to complex problems that require nestable skeleton patterns. The Calcium library contains SEQ, FARM, PIPE, IF,

FOR, WHILE, MAP, FORKE, and D&C skeletons[15].

The general idea of parallelism in Calcium is to store all tasks supplied by the programmers in a task pool. Tasks are then computed according to the skeleton instructions, and returned to the task pool again after completed.

QUAFF

QUAFF [86] is a C++ skeleton-based parallel programming library. It aims to reduce run-time overhead by using template-based meta-programming techniques.

The QUAFF library provides a number of nestable skeletons beside the constructs that allow these skeletons to be composed conditionally or sequentially in programs. Skeletons in the QUAFF library include a pipeline skeleton, farm skeleton for data parallelism, pardo skeleton, and scm skeleton which stand for the split-compute-merge model.

The implementation mechanism of QUAFF is to translate the C++ templates into new C+MPI code, which can be compiled and executed at run-time.

2.4.2.2 Multi-Core Computer Architectures

Since algorithmic skeletons were first introduced by Cole in [12], most structured parallel frameworks were proposed for distributed-memory environments such as clusters and grids (see section 2.4.2.1).

With the advent of multicore processor technology, several appropriate frameworks have been proposed recently.

To our knowledge, there are few frameworks that are introduced as a skeleton library to be implemented on shared-memory environments (in particular for multicore systems), this including Skandium [87] and FastFlow [51]. There are other non-skeleton frameworks such as TBB [50] that provide high level abstractions for low-level parallel activities in the same way as the algorithmic skeleton frameworks.

It is worthwhile to give a brief description of these non-skeleton frameworks. In the following section, we provide detailed description of available frameworks for multicore architectures.

All of the mentioned skeleton libraries address parallel or distributed systems and were not developed for use in grid systems.

Skandium

One of the recent skeleton libraries that supports shared-memory architectures (Multi-core system) is Skandium [87]. Skandium is a full Java library of shared memory algorithm skeletons. It is a complete re-implementation of the Calcium library described in Section 2.4.2.1.

Skandium is designed to exploit the strength of multicore systems by providing nestable skeletons for data and task parallelism. It contains all the algorithmic skeletons that exist in the Calcium library.

Skandium works using the Fork/Join framework in Java to achieve parallelism in shared memory environments, where all the operational details of the program

flow such as split, fork and join are implicitly defined in skeleton composition.

FastFlow

FastFlow [51, 88] is high-level parallel programming framework that target multi-core platforms. FastFlow is designed as a stack of layers, where the lowest layer provides very efficient lock-free synchronisation, the middle layer deals with the communication mechanisms, and finally the top layer provides as programming primitives.

FastFlow uses self-offloading technique to give the user an easy way to introduce the parallelism by offloading the kernels onto number of threads running on a multi-core CPU. Moreover, it allows users to move or copy parts of sequential codes into the body of C++ methods, for parallel execution in a FastFlow skeleton.

FastFlow provides programmers with a set of high-level parallel abstractions (algorithmic skeletons) as C++ template library. These abstractions include farm, farm-with-feedback (i.e. Divide&Conquer) and pipeline patterns.

TBB

Intel Threading Building Blocks (TBB)[50] is a commercial C++ library for shared-memory architecture. It is designed to be implemented on a multicore system by providing high level abstractions for parallel patterns.

Intel TBB supports data- and task parallel computations. It offers various

templates for parallel programming such as parallel_for, parallel_reduce, parallel_scan, and parallel_pipeline. These templates can be nested to build a large parallel components.

Additionally, Intel TBB provides concurrent data structures (containers) such as concurrent_hash_map, concurrent_queue, and concurrent_vector, beside a number of synchronisation primitives for multi-thread access.

TBB employs a work stealing technique to increase core utilisation which in turn improves the performance of the library implementations.

BlockLib

BlockLib[89] is a library of parallel generic building blocks for the IBM Cell processor that is used for gaming and high-performance computing.

BlockLib provides skeletons as compiled code and macros with the use of the C preprocessor and compiler. The library consists of data parallel skeletons such as map, reduce, scan, and map-with-overlap.

At the implementation level, memory management is controlled by using some functionality of the NestStep runtime system; and the synchronisation is controlled via the Cell signal [90].

2.4.2.3 Heterogeneous Environments

As we mentioned earlier, most of the algorithmic skeleton frameworks are introduced to take advantage of distributed-memory computing architectures where

each node in the system has a single-core CPU.

However, heterogeneous algorithmic skeletons are intended to efficiently take advantage of systems that comprise multicore CPU nodes. These heterogeneous skeletons are based on both distributed- and shared-memory programming models to support the heterogeneous environment.

In this section, we provide a brief description of algorithmic skeletons framework for heterogeneous platforms.

Extension of Muesli

An extension of the Muesli library described in Section 2.4.2.1 is presented in [16] for multicore computer architectures. It combines OpenMP and MPI to efficiently exploit multicore clusters, where the parallelism within each node is achieved via OpenMP.

The new version includes all the distributed data structures(arrays, matrices) and their supported skeletons(fold, map, scan, zip).

Therefore, the enhanced version of the Muesli library provides more flexibility and implementations by supporting both multi-node, single-core and multicore architectures.

SkeTo

SkeTo described in Section 2.4.2.1 is a parallel skeletons library which was originally proposed for distributed environments. A new implementation of SkeTo is

presented in [17] to support multicore clusters environments.

The new implementation provides new parallel skeletons such as generate, map, and reduce for distributed matrices in the SkeTo skeleton library.

To ensure load-balancing between nodes and cores, a two-stage dynamic task scheduling strategy is employed with the new skeletons. Task scheduling among cores is implemented by dividing the task into smaller tasks using the size of the L1 cache,and then applying the task-steal strategy for task allocation. Between the nodes the task is divided according to the size of the L2 cache by using a master-worker model.

$\mathbf{nmc}_{muskel}$

nmc_{muskel} (or networked multicore muskel) [91] is a new version of the Muskel framework described in Section 2.4.2.1. The new version provides a set of parallel skeletons for implementation on multicore clusters. nmc_{muskel} implements the same set of parallel skeletons as the muskel library.

llc Language

llc [60, 92] is a high-level parallel language that offers parallel algorithmic skeletons for writing parallel programs. Parallel skeletons are provided using OpenMP directives in the language syntax along with a source to source compiler. llc uses C as the target language.

llc provides support for algorithmic skeletons that can be executed on distributed or shared memory architectures. The available skeletons in llc include a forall skeleton, parallel sections, task farms and pipelines.

In the llc language, parallelism is achieved using MPI on both distributed and shared memory architectures. Thus, llc programs are written in C with use of llc parallel constructs, and then the compiler translates the code to C with MPI calls.

A new methodology in [93] takes advantage of the llc compiler to generate hybrid MPI/OpenMP code for multicore architectures, where OpenMP is used inside each node in the cluster and MPI controls the communications between these nodes.

2.4.2.4 *Skeletal*-based GPU Programming

A number of data parallel programming languages have been introduced for GPGPU programming. CUDA, and OpenCL are widely used as discussed in the previous sections. To gain performance most GPGPU languages, including both CUDA and OpenCL, are low level and lack high level abstractions to improve programmability.

A number of approaches have been proposed to improve and ease GPU programming. For example, a compiler framework by Baskaran et al.[94] has been introduced for automatic transformation of sequential input programs into efficient parallel CUDA programs. Another compiler framework by Seyong et al.[95]

aims to translate standard OpenMP applications into CUDA-based GPGPU applications to offer an easier programming model for GPGPU computing.

Eventually, the skeleton parallel programming approach, which has been shown to deliver significant high performance on general-purpose CPU architectures, fills the gap and abstracts the GPU infrastructure from the purpose of the program. With the advent of the CUDA programming model and the OpenCL standard, much research [96, 97, 98, 99, 11] has been done in the area of GPU skeletal-based programming. Even more, using parallel skeleton algorithmic for both CPU and GPU can provide multiple implementations on heterogeneous multicore/GPU systems. In the following sub-sections, we provide a brief description of existing parallel algorithmic skeleton frameworks that target GPU computing.

Thrust

Thrust[96] is an open source C++ library that uses CUDA on NVIDIA GPU architectures. It is based on the Standard Template Library (STL) to provide vector type (generic containers) for memory management in both the host CPU and the device GPU.

Thrust provides a high-level parallel programming environment through a collection of high-level data-parallel primitives such as scan, sort, and reduce. Even more, in the Thrust library, a complex algorithm can be implemented by composing the provided parallel primitives together.

CUDPP

CUDPP[97] is a library of high-level primitives similar to skeletons. It is implemented in C/C++ using CUDA on NVIDIA GPU architectures. CUDPP offers data-parallel primitives such as parallel scan, parallel sort, and parallel reduction. These primitives can be used as building blocks for a wide variety of data-parallel algorithms, and also can be used as standalone calls on the GPU.

SkePU

SkePU[98, 100] is a C++ template library which offers a set of data-parallel skeletons for GPU computing. The SkePU library is built on top of CUDA and the OpenCL standard to provide single- and multi-GPU programming environment to non-graphics developers. The library also supports multicore CPUs programming by using OpenMP.

SkePU provides flexible memory management by using a vector data container that implements the lazy memory copying technique to avoid unnecessary memory transfer. The set of data-parallel skeletons implemented in SkePU library includes the basic data-parallel Map, Reduce and Scan skeletons, and other variants of the map skeleton such as MapOverlap and MapArray.

SkePU uses preprocessor macros to provides user-defined functions that can be used with skeletons; these functions are implemented as structs with member functions for CUDA and CPU and strings for OpenCL.

SkelCL

SkelCL [99] is a skeleton library for high-level GPU programming. Like SkePU, SkelCL is implemented using the OpenCL standard to ensure hardware portability for a wide range of heterogeneous systems. SkelCL offers support for programs on multiple devices, in particular for multi-GPUs.

Currently, the library provides a set of basic data-parallel skeletons including Map, Zip, Reduce, and Scan. Beside these algorithmic skeletons, it provides an abstract vector data type for memory management such as transferring data between multiple compute devices (CPU and GPU). All the skeletons in SkelCL use this vector as input and output in their implementations.

Qilin

Qilin [11] is a high-level parallel programming model that provides an API that implements common data-parallel operations to exploit hardware parallelism available on heterogeneous architectures. The Qilin API built on top of C/C++ with the use of Intel Thread Building Blocks [50] for CPU programming and NVIDIA CUDA [1] for GPU programming. Both TBB and CUDA codes are generated during compilation using the Qilin compiler and then the final native machine code is generated using the system compiler. Current implementation of Qilin provides two C++ templates (*QArray*, *QarrayList*) for parallel computations. *QArray* represents a multidimensional array of a generic type, while *QarrayList*

represents a list of *QArray* objects.

Muesli

An extension of the Muesli library (described in sections 2.4.2.1, 2.4.2.3) is presented in [101] to provides algorithmic skeletons for GPU implementations. The extension library provides support to a variety of parallel architectures including multi-core, multi-GPU, and GPU clusters [102]. It uses MPI for distributed memory architectures, OpenMP for multi-core, and CUDA for GPU programming.

FastFlow

FastFlow has indirect support for GPU implementations through GPU-enable linear algebra libraries [103]. A Heterogeneous streaming pipline implementation using the FastFlow library for large scale computational problem is provided to support parallel implementations on multi-core CPUs and multi-GPUs in a cluster environment.

2.5 Discussion

Table 1 summaries the characteristics of well-known parallel skeletal-based frameworks and their implementations. One can see that there is not much research in the area of heterogeneous parallel skeletons that target heterogeneous parallel

architectures such as heterogeneous multicore cluster and heterogeneous multicore/GPU cluster. The development of skeleton-based parallel programming libraries, which will be presented in the following chapter, has been inspired by a number of similar frameworks outlined in Section 2.4.2.3 to provide heterogeneous skeletons for heterogeneous parallel architectures.

However, the implementation philosophy of our libraries differ from these frameworks in that the skeletons can be executed on a variety of parallel architectures (See figure 5) in a transparent way. This means the programmer does not need to choose the appropriate skeleton for the target hardware; instead the skeleton automatically implements a suitable model for the specific heterogeneous multicore cluster architecture.

Another difference is that the target hardware of approach is a heterogeneous parallel system composed of a number of nodes either with different processing capability or with an integrated multicore/GPU architecture. Thus, performance cost models (presented in Chapter 4 and 6) are integrated into our heterogeneous skeletons to balance workload distribution to improve the performance on heterogeneous architectures.

Framework	Host Language	Execution Environment	GPU Implementation	Multi-GPU	skeleton set
P3L	C	Distributed	NO	NO	farm, pipeline, seq, map, reduce, comp
Skil	C subset	Distributed	NO	NO	array-permute-rows, array-fold, array-gen-mult, array-map, array-gen-mult, array-permute-rows
HDC	Haskell	Distributed	NO	NO	dcA, dcB, dcC, dcD, dcE,dcF
Muesli	C++	Distributed & multicore/GPU	CUDA	YES	Pipeline, Farm, DivideAndConquer, BranchAndBound, fold, map,zip
Lithium	Java	Distributed	NO	NO	pipe, farm, loop, while, map, DivideConquer, if
eSkel	C	Distributed	NO	NO	Pipeline, Deal, Farm, HaloSwap, Butterfly
Muskel	Java	Distributed	NO	NO	farm, pipeline, map
SkeTo	C++	Distributed & Het.multicore	NO	NO	map, reduce, scan
Calcium	Java	Distributed	NO	NO	SEQ, FARM, PIPE, IF, FOR, WHILE, MAP,FORKE,D&C
QUAFF	C++	Distributed	NO	NO	pipeline, farm, pardo, scm
Skandium	Java	Multi-Core	NO	NO	SEQ, FARM, PIPE, IF, FOR, WHILE, MAP,FORKE,D&C
FastFlow	C++	Multi-Core	NO	NO	farm, farm-with-feedback, pipeline
TBB	C++	Multi-Core	NO	NO	parallel_for, parallel_reduce, parallel_scan, parallel_pipeline
BlockLib	C	IBM Cell processor	NO	NO	map, reduce, scan, map_with_overlap
nmc_{muskel}	Java	Het.multicore	NO	NO	farm, pipeline, map
llc	C	Het.multicore	NO	NO	forall, farm, pipeline
Thrust	C++	GPU	CUDA	NO	Scan, Sort, Reduce
CUPP	C/C++	GPU	CUDA	NO	Scan, Sort, Reduce
SkePU	C++	CPU/GPU	CUDA/OpenCL	YES	map, reduce, MapReduce, MapArray, MapOverlap
SkelCL	C++	GPU	OpenCL	YES	Map, Zip, Reduce, Scan
Qilin	C/C++	GPU	CUDA	NO	QArray, QarrayList

Table 1: Algorithmic skeleton frameworks characteristics

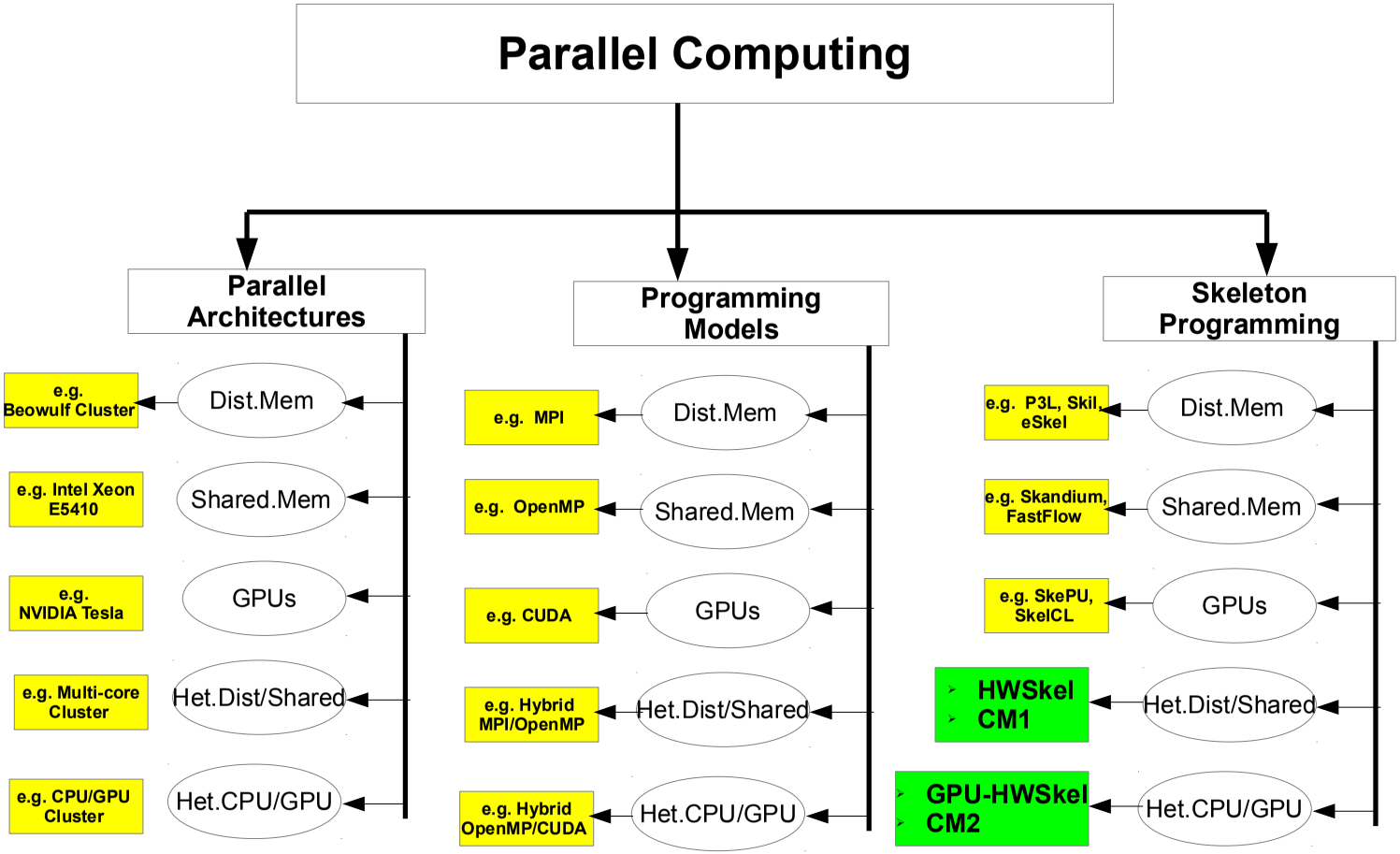


Figure 5: Parallel Taxonomy of HWSkel and GPU-HWSkel Frameworks

Chapter 3

The *HWSkel* Library

This chapter presents a C based-skeleton library that uses MPI and OpenMP to achieve parallelism on heterogeneous multicore cluster. The chapter starts by providing a justification for using C as the target language, and also the choice of MPI and OpenMP for parallel implementations in Section 3.1. Next we provide a description of each skeleton in the current *HWSkel* prototype in Section 3.2.

3.1 C skeleton-based Library

In this section we first discus the design of the *HWSkel* library and its parallel implementations. We then show how to take advantage of MPI and OpenMP to achieve parallelism on both distributed and shared memory architectures in heterogeneous multicore systems.

3.1.1 Design Summary

Our design goals are as follows. We aim to provide a high-level heterogeneous programming model that can hide many low-level details that are commonly encountered in any parallel application on heterogeneous parallel architectures. Thus programmers need only concentrate on the key application-specific issues.

The skeletons in *HWSkel* are implemented using a hybrid OpenMP/MPI model. As discussed in Section 1.2.2, our framework enables the programmer to develop parallel programs in the C language in a sequential manner, where the skeleton call appears as a normal function call in the program.

This design is adaptable and hence *HWSkel* skeletons can be used for distributed-memory systems, shared-memory systems or both systems together as heterogeneous multicore systems. For instance, if the underlying system is distributed memory, the distributed parallel programming model will be automatically adopted. The *HWSkel* library has the following characteristics:

1. The recent trend of designing algorithm skeletons is to present them as libraries to avoid adding any new syntax. Therefore, *HWSkel* is provided as a library for C that works using MPI and OpenMP to achieve the parallelisation on heterogeneous multicore systems.

2. The *HWSkel* library provides simple heterogeneous parallel skeletons for data parallelism for heterogeneous multicore clusters.

3. *HWSkel* supports parallelism on heterogeneous multicore architectures, and

flexible parallelism on both shared and distributed memory architectures.

4. Lower-level details of parallel programming are concealed from the users by our skeleton. Furthermore, the interaction between MPI and OpenMP introduces new communication such as data flow between these models, and this communication is implicitly defined by skeleton composition. Hence skeleton can be used to develop parallel programs in a sequential fashion.

5. To ensure a good load balance we integrate an effective cost model (CM1) for data-load distribution into our system as discussed in Chapter 4. The cost model uses specific hardware properties to distribute work between processors.

3.1.2 Cole's Manifesto

In our framework we have adopted Cole's four manifesto principles[64] as a guide to design the *HWSkel* library. In this section, we show how the characteristics of our skeletons satisfy these principles.

- ✔ ***Propagate The Concept with Minimal Conceptual Disruption.*** To satisfy this principle *HWSkel* is provided as a library of C functions on top of MPI and OpenMP to avoid the introduction of any new syntax.

- ✔ ***Integrate ad-hoc Parallelism.*** Since *HWSkel* runs on top of most popular parallel programming models such as MPI and OpenMP, the integration with ad-hoc parallelism is facilitated at this level.

- ✔ ***Accommodate Diversity***. *HWSkel* skeletons cannot be directly nested. However, in our library, heterogeneity is realised internally by the layered composition of architecture specific skeleton components so we anticipate that it should be straight forward to expose this to programmers. Thus we satisfy the requirement for diversity.

- ✔ ***Show the payback***. To show the payback we build our skeletons around the idea of supporting different architectures. Since we have used two different parallel programming models in the same skeleton, skeletal programs can be executed on different architectures.

3.1.3 Host Language

From a programmer's perspective, providing parallel skeletons as high-level libraries in an existing language is more common than the new language approach. This is due to programmer reluctance to learn a new programming language. Thus we introduce our approach as a high-level library of parallel algorithmic skeletons, which can be used by C programmers in as normal function call in the program. Our library is written in C due to the popularity of imperative languages in the parallel domain. In addition C has the following important features that make it a good candidate host language for algorithmic skeletons.

- **Higher-order functions:** typically algorithmic skeletons are offered to programmers as higher-order functions, due to their natural properties [104]

that make them highly abstract, where the details of a problem are passed as arguments. Thus, skeletons can easily be implemented in C since it supports function pointers as function parameters. Thus skeletons can imitate higher-order functions by accessing arbitrary argument functions. However, this does not support function arguments as closures.

- **Ability to write polymorphic functions:** since the idea of the algorithmic skeleton approach is to encapsulate common parallel patterns into polymorphic higher-order functions [105], therefore, generic skeletons can be provided in C by using void pointers.

3.2 Algorithmic Skeletons in *HWSkel*

This section presents the current prototype of the *HWSkel* library that includes heterogeneous algorithmic skeletons for data parallel computations. We provide a brief description and definition for each heterogeneous skeleton in the *HWSkel* library. Note that all the definitions of the skeletons are written in BMF [106].

The current *HWSkel* library defines some widely used skeletons to provide a baseline for comparison with other people's work. The skeletons include *hMap*, *hMapAll*, *hReduce*, *hMapReduce*, and *hMapReduceAll* for data parallelisation, which are based on distributed data structures.

3.2.1 Data Communication

All our data-parallel skeletons work on homogeneous data structures (mostly arrays), so dealing with heterogeneous data structures has to be in a high level way by mapping them to homogeneous data structures. This is done by means of the *ArrayList*, where each element contains a buffer of *char* data type and the length of the buffer. So the data structures need to be packed into this *ArrayList* before the distribution process and then unpacked on each processor.

3.2.2 Initialisation and Termination

Since the implementations of the *HWSkel* communication system uses MPI at the top level, the skeletons must be invoked after the initialisation of the MPI environment. For this reason we defined two wrapper functions to initiate and terminate the MPI environment. In principle, these could be added automatically by a pre-processor.

3.2.2.1 *InitHWSkel*()

The skeletons that are defined in the *HWSkel* library must be called after *InitHWSkel*. *InitHWSkel* initialises the MPI environment and the global ArrayList **currentCluster** that contains all the architectural information that is needed in the CM1 cost model as discussed in Section 4.4. It calls the *getClusterInfo()* function that does dynamic parametrisation of the static cost model (CM1), as discussed in Section 4.5.

```
1  void InitHWSkel(int argc, char **argv)
2  {
3     //- MPI Initialisation
4     MPI_Init (&argc, &argv);
5     MPI_Comm_rank (MPI_COMM_WORLD, &id);
6     MPI_Comm_size (MPI_COMM_WORLD, &np);
7
8     //- get cluster specifications
9     currentCluster = GetClusterInfo();
10 }
```

Listing 3.1: InitHWSkel Code.

The prototype for the *InitHWSkel* is:

```
void* InitHWSkel(int argc,char **argv)
```

Listing 3.1 shows C code of *InitHWSkel* C code. A complete code for the *getClusterInfo()* function can be found in Appendix B.3.

3.2.2.2 *TerminateHWSkel*()

The *TerminateHWSkel* function is called to terminate a computation based on skeletons.

The prototype for the *TerminateHWSkel* is:

```
void* TerminateHWSkel()
```

3.2.3 The *hMap* Skeleton

The *HWSkel* library provides a heterogeneous map skeleton called *hMap*. *hMap* is equivalent to the classical map skeleton for data-parallel computations, where a single function is applied to different data elements of input data structures (usually an array). To achieve parallelism, a collection of input data structures

are distributed amongst a group of processing elements; then the map function is applied to each data element in parallel and then the results are collected.

hMap is implemented using a Single-Program-Multiple-Data programming model, where the input data structures and the results are contained in a single node (usually the node with rank 0). Since the underlying target hardware consists of two levels of parallel hardware architecture, *hMap* first partitions and distributes the input data structures amongst the nodes, and then partitions the local data amongst the CPU cores in each node in the system. After the map function is executed by each processing element the local results are collected in each node, and then all results are gathered by the master node.

hMap BMF Definition

The *hMap* skeleton applies function f to each element in array *a[]*.

$$hMap(f, a[]) = [f(a[0]), f(a[1]), ..., f(a[n-1])] \quad (1)$$

hMap Interface:

The prototype for *hMap* is:

```
void* hMap(void* dataList,int size,enum DataType dType,
        void* mapFunc, enum DataType rType);
```

where

`dataList` Specifies the starting address of the data.

`size`	Indicates the length of the data.
`dType`	Denotes the datatype of input data.
`mapFunc`	Specifies the map function.
`rType`	Denotes the datatype of the output data.

***hMap* Algorithm:**

Algorithm 1 displays the implementation of the SPMD model in the *hMap* skeleton. Complete code for *hMap* can be found in Appendix C.1.

Algorithm 1 hMap skeleton implementation

```
1: BEGIN
2: → master node:
3: for every node_i do in parallel
4:    Send chunks-list[i]
5: end for
6: → worker node:
7: for every core_i do in parallel
8:    assign core-local-list from node-chunks-list[i]
9: end for
10: → each core:
11: for every core-local-list[i] do in parallel
12:    core-local-results[i] = map( f , core-local-list[i])
13: end for
14: → worker node:
15: for every core_i do in parallel
16:    concatenate (core-local-results[i], node-results[i])
17: end for
18: → master node:
19: for every node_i do in parallel
20:    concatenate (node-results[i], global-results[i])
21: end for
22: END
```

hMap Example

An example of using *hMap* to calculate the element-wise square is shown in Listing 3.2

3.2.4 The *hMapAll* Skeleton

The *hMapAll* Skeleton is similar to *hMap*, but all input data is sent to each processing element in the system, and then each gets its own portion.

hMapAll BMF Definition

The *hMapAll* skeleton applies function f to each element in array *a[]* against the whole *a[]*.

$$hMapAll(f, a[]) = [f(a[0], a[]), f(a[1], a[]), ..., f(a[n-1], a[])] \quad (2)$$

hMapAll Interface:

The prototype for *hMapAll* is:

```
void* hMapAll(void* dataList,int size,enum DataType dType,
        void* mapFunc, enum DataType rType);
```

where

`dataList` Specifies the starting address of the data.

`size` Indicates the length of the data.

`dType` Denotes the datatype of input data.

```
1
2 #include "HwSkel.h"
3
4 int square(int a)
5 {
6    return a * a;
7 }
8
9 int main(int argc, char **argv)
10 {
11     int len;
12     int *input;
13     int *output;
14
15     //- skeleton initialization
16     InitHwSkel(argc,argv);
17
18     if(StartNode)
19     {
20        sscanf(argv[1],"%d",&len);
21
22        //- allocate memory for an array
23        input = malloc(len*sizeof(int));
24
25        //- fill the array
26        int i;
27        for(i=0;i<len;i++)
28           input[i] = i;
29      }
30
31     //- call hMap skeleton
32     output = hMap(input, length, INT, square);
33
34     if(StartNode)
35     {
36        for(i=0;i<len;i++)
37            printf(" parallel result = %d \n", output[i]);
38     }
39
40     //- skeleton termination
41     TerminateHwSkel();
42
43     return 0;
44 }
```

Listing 3.2: A hMap example that calculate the element-wise square.

`mapFunc` Specifies the map function.

`rType` Denotes the datatype of the output data.

hMapAll Algorithm:

Algorithm 2 displays the implementation the SPMD model in *hMapAll*. Complete code for *hMapAll* can be found in Appendix A.2.

Algorithm 2 hMapAll skeleton implementation

```
1: BEGIN
2: → master node:
3: for every node_i do in parallel
4:    Send input-data
5: end for
6: → worker node:
7: for every core_i do in parallel
8:    assign core-local-list from input-data
9: end for
10: → each core:
11: for every core-local-list[i] do in parallel
12:    core-local-results[i] = map( f , core-local-list[i], input-data)
13: end for
14: → worker node:
15: for every core_i do in parallel
16:    concatenate (core-local-results[i], node-results[i])
17: end for
18: → master node:
19: for every node_i do in parallel
20:    concatenate (node-results[i], global-results[i])
21: end for
22: END
```

hMapAll Example

Listing 3.3 shows an example of *hMapAll* that find the frequency of array elements.

```
1  #include "HwSkel.h"
2
3  int frequency(int a, int *arr, int len)
4  {
5     int i, freq = 0;
6     for(i=0;i<len;i++)
7        if(a == arr[i])
8           freq++;
9      return freq;
10 }
11
12 int main(int argc, char **argv)
13 {
14     int len;
15     int *input, *output;
16
17     //- skeleton initialization
18     Init_HwSkel(argc,argv);
19
20     if(StartNode)
21     {
22        sscanf(argv[1],"%d",&len);
23
24        //- allocate memory for an array
25        input = malloc(len*sizeof(int));
26
27        //- fill the array
28        int i;
29        for(i=0;i<len;i++)
30           input[i] = rand() % 100;
31     }
32
33     //- call hMapAll skeleton
34     output = hMapAll(input, length, INT, frequency);
35
36     if(StartNode)
37     {
38        for(i=0;i<len;i++)
39           printf(" parallel result = %d \n", output[i]);
40     }
41
42     //- skeleton termination
43     Terminate_HwSkel();
44     return 0;
45 }
```

Listing 3.3: A hMapAll example that finds the frequency of array elements.

3.2.5 The *hReduce* Skeleton

The *hReduce* skeleton represents the reduce function, where all elements in input data structures (usually arrays) are "summed-up" using an associative binary function. As in *hMap* the input data structures are partitioned and distributed to all nodes in the system, and then the local input data for each node is split amongst the cores. Finally the *reduce* function firstly merges all the local intermediate results, and then perform the reduction on the global results.

hReduce BMF Definition

The *hReduce* function converts an array of numbers to a single value using an associative binary operator $\oplus$.

$$hReduce(\oplus, a[]) = a[0] \oplus a[1] \oplus ..... \oplus a[n-1] \tag{3}$$

hReduce Interface:

The prototype for the *hMapReduce* skeleton is:

```
void* hReduce(void* dataList,int size,enum DataType dType,
        void* reduceFunc, enum DataType rType);
```

where

`dataList` Specifies the starting address of the data.

`size` Indicates the length of the data.

`dType` Denotes the datatype of input data.

`reduceFunc` Specifies the reduction function.

`rType` Denotes the datatype of the output data.

***hReduce* Algorithm:**

Algorithm 3 display the implementation of the SPMD model in *hReduce*. Complete code for *hReduce* can be found in Appendix A.3.

Algorithm 3 hReduce skeleton implementation

```
1: BEGIN
2: → master node:
3: for every node_i do in parallel
4:    Send chunks-list[i]
5: end for
6: → worker node:
7: for every core_i do in parallel
8:    assign core-local-list From node-chunks-list[i]
9: end for
10: → each core:
11: for every core-local-list do in parallel
12:    core-result = merge( ⊕ , core-local-list )
13: end for
14: → worker node:
15: for every core_i do in parallel
16:    node-result = merge (⊕, core-results[i])
17: end for
18: → master node:
19: for every node_i do in parallel
20:    result = merge (⊕, node-results[i])
21: end for
22: END
```

***hReduce* Example**

Listing 3.4 shows an example of *hReduce*, which applies reduction computation by using + as operator.

```
1
2 #include "HwSkel.h"
3
4 int plus(int a, int b)
5 {
6     return a + b;
7 }
8
9 int main(int argc, char **argv)
10 {
11     int len;
12     int *input;
13     int *output;
14
15     //- skeleton initialization
16     Init_HwSkel(argc,argv);
17
18     if(StartNode)
19     {
20         sscanf(argv[1],"%d",&len);
21
22         //- allocate memory for an array
23         input = malloc(len*sizeof(int));
24
25         //- fill the array
26         int i;
27         for(i=0;i<len;i++)
28            input[i] = i;
29     }
30
31     //- call hReduce skeleton
32     output = hReduce(input, length, INT, plus);
33
34     if(StartNode)
35     {
36        printf(" parallel result = %d \n", output);
37     }
38
39     //- skeleton termination
40     Terminate_HwSkel();
41
42     return 0;
43 }
```

Listing 3.4: A hReduce example that applies reduction computation by using + as operator.

3.2.6 The *hMapReduce* Skeleton

The *hMapReduce* skeleton is built from the basic data parallel skeletons: *map*, *reduce*, and *split*. The underlying conceptual model is similar to Google's MapReduce [107] but the target architectures are heterogeneous multicore clusters. The computation in the *hMapReduce* skeleton is expressed as two functions: a *map* function that processes the input data and generates an array of intermediate results, and a *reduce* function that merges all the intermediate results into a single result. Here the *map* function generates a local array of intermediate results within each multicore node, the *reduce* function firstly merges all the local intermediate results, and then perform the reduction on the global results.

hMapReduce BMF Definition

Simply, the *hMapReduce* skeleton is a generalisation of the map skeleton including the reduce skeleton.

$$hMapReduce(f, \oplus, a[]) = reduce(\oplus, map(f, a[])) \tag{4}$$

So from the equations (1), (3), and (4) we can rewrite the *hMapReduce* skeleton as follows:

$$hMapReduce(f, \oplus, a[]) = f(a[0]) \oplus f(a[1]) \oplus ... \oplus f(a[n-1]) \tag{5}$$

where *a[]* is an array of elements, f is a function applied to each element in *a[]*, and $\oplus$ is an associative operation.

The *hMapReduce* Interface:

The prototype for *hMapReduce* is:

```
void* hMapReduce(void* dataList,int size,enum DataType dType,
        void* mapFunc, enum DataType rType, void* reduceFunc);
```

where

`dataList`	Specifies the starting address of the data.
`size`	Indicates the length of the data.
`dType`	Denotes the datatype of the input data.
`mapFunc`	Specifies the map function.
`rType`	Denotes the datatype of the output data.
`reduceFunc`	Specifies the reduction function.

hMapReduce Algorithm:

Figure 6 shows the computation scheme for the *hMapReduce* skeleton. The skeleton employs a SPMD programming model that is inherited from MPI, the top level of the implementation of the *hMapReduce* skeleton. Complete code for *hMapReduce* can be found in Appendix A.4.

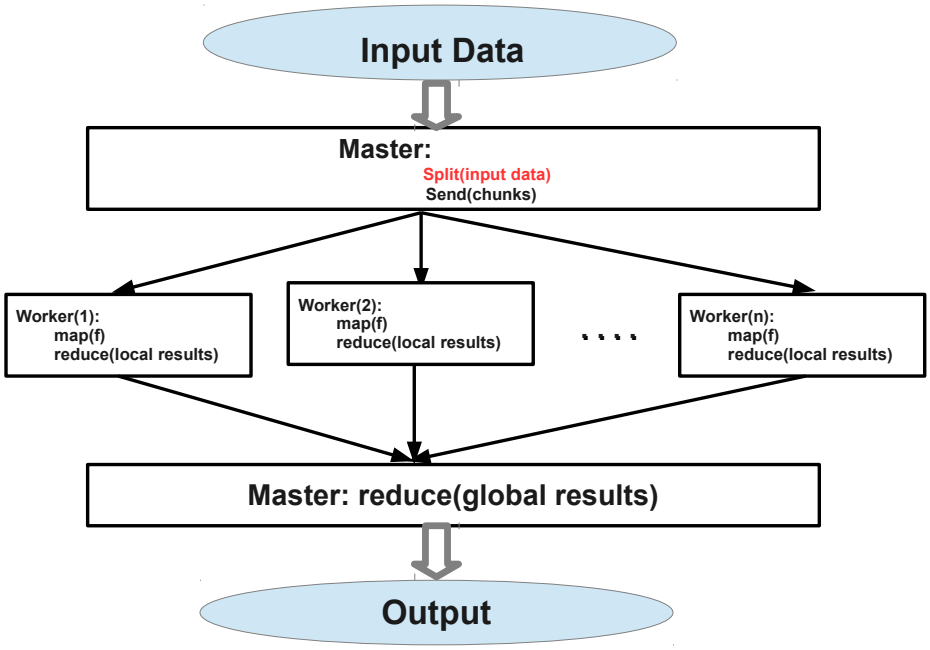


Figure 6: The Computation Scheme for *hMapReduce* Skeleton

***hMapReduce* Example**

Listing 3.5 shows an example of *hMapReduce* that computes the vector dot product.

3.2.7 The *hMapReduceAll* Skeleton

The *hMapReduceAll* skeleton is similar *hMapReduce* described above, but all input data is sent to each processing element in the system, and then each processing element splits the data to get its own portion. Thus, both the input data and the portion of the input data in each node can be used together by the *map* function to perform its computation.

```
1
2 #include "HwSkel.h"
3
4 int square(int a)
5 {
6   return a * a;
7 }
8
9 int plus(int a, int b)
10 {
11   return a + b;
12 }
13
14 int main(int argc, char **argv)
15 {
16   int len;
17   int *input, *output;
18
19     //- skeleton initialization
20     Init_HwSkel(argc,argv);
21
22     if(StartNode)
23     {
24      sscanf(argv[1],"%d",&len);
25
26      //- allocate memory for an array
27      input = malloc(len*sizeof(int));
28
29      //- fill the array
30      int i;
31      for(i=0;i<len;i++)
32         input[i] = i;
33     }
34
35     //- call hMapReduce skeleton
36     output = hMapReduce(input, length, INT, square, INT, plus);
37
38     if(StartNode)
39     {
40        printf(" parallel result = %d \n", output);
41      }
42     //- skeleton termination
43     Terminate_HwSkel();
44     return 0;
45 }
```

Listing 3.5: A hMapReduce example that compute the dot product.

hMapReduceAll BMF Definition

$$hMapReduceAll(f, \oplus, a[]) = f(a[0], a[]) \oplus f(a[1], a[]) \oplus ... \oplus f(a[n-1], a[]) \quad (6)$$

where *a[]* is an array of elements, f is a function applied to each element in *a[]* against the all elements in *a[]*, and $\oplus$ is an associative operation.

The *hMapReduceAll* Interface:

The *hMapReduceAll* skeleton has the same for prototype as *hMapReduce*, but the implementation is different:

```
void* hMapReduce(void* dataList,int size,enum DataType dType,
          void* mapFunc, enum DataType rType, void* reduceFunc);
```

where

`dataList`	Specifies the starting address of the data.
`size`	Indicates the length of the data.
`dType`	Denotes the datatype of the input data.
`mapFunc`	Specifies the map function.
`rType`	Denotes the datatype of the output data.
`reduceFunc`	Specifies the reduction function.

hMapReduceAll Algorithm:

Figure 7 shows the computation scheme for *hMapReduceAll*.

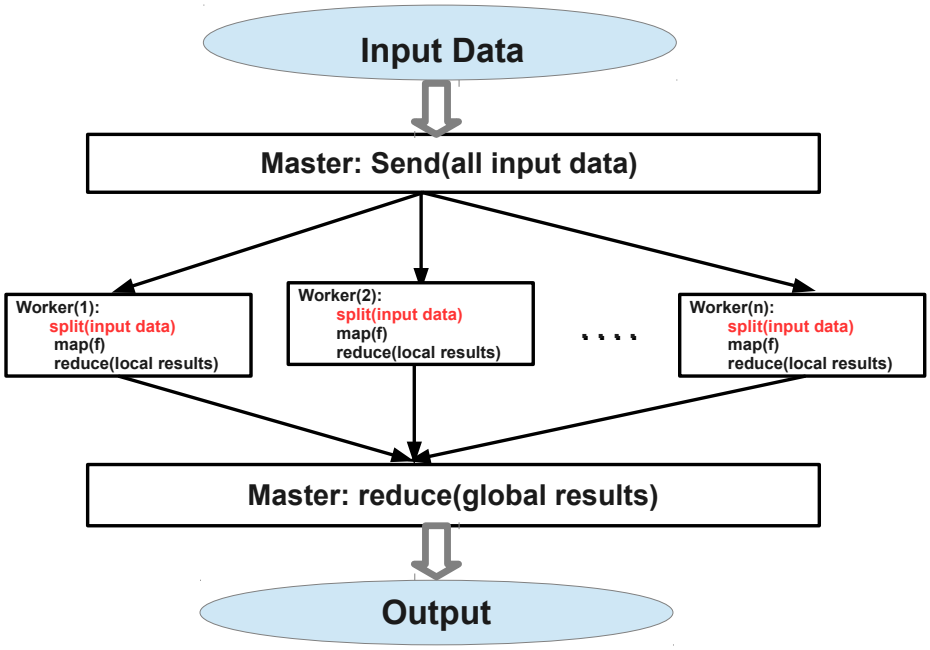


Figure 7: The Computation Scheme for *hMapReduceAll* Skeleton

3.3 Summary

In this chapter, we have presented a new skeleton-based programming library called *HWSkel* for heterogeneous systems, in particular, heterogeneous multicore cluster architectures.

The library is implemented in C on top of MPI as a distributed-memory programming model and OpenMP for shared-memory parallelism. This means that the heterogeneous skeletons can take straightforward advantage of their underlying hybrid programming model to execute on either distributed-memory systems, shared-memory systems or distributed-shared memory architecture.

In particular, the *HWSkel* framework provides a set of heterogeneous skeletons

for data-parallel computations i.e. *hMap, hMapAll, hReduce, hMapReduce, and hMapReduceAll.* Moreover, since our skeletons need to be invoked within an MPI initialisation, the *HWSkel* library provides wrapper functions (*e.g InitHWSkel and TerminateHWSkel*) for some MPI routines to keep the programmer away from using a new programming language within the skeletal programs .

Chapter 4

The *HWSkel* Cost Model (CM1)

This chapter presents a performance cost model (CM1) for data parallelism on heterogeneous multicore cluster architectures. The aim of CM1 is to provide a cost estimation of a given problem on a given machine by using a number of hardware properties, for use it in our heterogeneous skeletons presented in Section 3.2. We start by providing a survey of several important parallel cost models in Section 4.1 and then discuss the resource metrics of some current cost models in Section 4.2 . In sections 4.3 and 4.4 we introduce our new architecture-aware cost model and then illustrate its use in the *HWSkel* library in Section 4.5. Finally, we discuss experimental results to evaluate the effectiveness of our new cost model in the performance of the *HWSkel* heterogeneous skeletons with both data parallel benchmark and a non-trivial image analysis application in Section 4.6 .

4.1 High-Level Parallel Cost Models

Models of parallel computation play an important role in designing and optimising parallel algorithms and applications. These models assist the developer in understanding all important aspects of the underlying architecture without knowing unnecessary details. Moreover, parallel computational models are used to predict the performance of a given parallel program on a given parallel machine.

The common way of predicting the performance of parallel program is to derive a symbolic mathematical formula that describes the execution time of that program. This formula has a set of parameters which usually include the size of program, number of processors, and other hardware and algorithm characteristics that affect the execution time of the program. These parameters can be given by a programmer, benchmarking, or profiling tools.

Skeleton-based and similarly structured frameworks have employed these parallel computational models to predict the performance of the parallel algorithms in the early stages of the design process. Consequently, these computational models can assist and guide scheduling algorithmic skeletons on a wide variety of architectures.

Several parallel computational models have been developed for parallel distributed systems to guide parallel algorithm designers. Good general surveys of early research are given in [108, 109, 110, 13] and a more recent survey is given in [111].

In this section, we survey several well-known parallel cost models that have been proposed for parallel and distributed environments as well as algorithmic skeletons.

4.1.1 The Family of PRAM Models

The most widely-used cost model in parallel computing is the Parallel Random Access Machine (PRAM) model[112]. The PRAM model is based on the RAM model[113] of sequential computation. The model consists of a global shared memory and a set of sequential processors that operate synchronously. The model makes the assumption that at each synchronous step, each processor can access any memory location in one unit time regardless of the memory location. The PRAM model provides a useful guide for parallel algorithm designers and thereby allows them to ignore all the architecture details of the underlying hardware and concentrate on application-specic issues.

Despite the useful basis provided by the PRAM model for parallel algorithm design, it can not reflect all the costs of a real parallel machine. This results in non-portable programs due to a number of assumptions made by the model by ignoring the cost of some parallel activities such as synchronisation, memory contention, and communication latency or bandwidth.

Therefore, several realistic variants of PRAM-based models have been introduced to make PRAM more practical. These variants attempt to account for the cost issues of real parallel machines. For example, models such as Block

PRAM(BPRAM)[114], Local-Memory PRAM(LPRAM)[115], and Asynchronous PRAMs[116] seek to include the latency cost with the standard PRAM model.

Another PRAM variant are asynchronous PRAMs that add some degree of asynchrony into the basic PRAM model in order to ease the restriction on processors synchronisation. These models differ in the way of the processors are synchronised. They include the Asynchronous Parallel Random Access Machine model(APRAM)[117] that addresses the synchronisation assumption of the basic PRAM model to allow asynchronous execution, and the Hierarchical PRAM(H-PRAM) model[118], which use the PRAM model as a sub-model, consists of a collection of synchronous PRAMs that operate asynchronously from each other. Another asynchronous model by Gibbons et al. [119] allows the processors to run in an asynchronous manner.

The CRCW PRAM model [120] and the QRQW PRAM model [121, 122] account for memory locations contention, where the read and write to shared memory locations are done concurrently.

4.1.2 BSP and Variants

The Bulk Synchronous Parallel (BSP) model [123, 124] is a parallel computation model that provide a simple way of writing parallel programs for a wide-range (architecture-independent) of parallel architectures by offering a bridging model that links software and architecture. Also it provides a straightforward way

for realistic performance prediction for application design on a variety of different parallel architectures including distributed-memory systems, shared-memory multiprocessors, and networks of workstations. Practically, the BSP model aims to provide a bridge model between the software and hardware.

The BSP model consist of a collection of processors that communicate using message passing. The computations in the BSP model are formulated as a series of *supersteps*. Conceptually, each superstep is divided into three stages. In the first stage, all processors concurrently compute using only local data. In the second stage, processors exchange messages with each other. In the third stage, all of the processors execute a barrier synchronisation, after they finished sending and receiving messages.

Compared with the PRAM model described in Section 4.1.1, the BSP model is more realistic, since it accounts for two cost issues of the real parallel machines, namely communication cost and memory latency cost.

Since BSP programs are based on sequential supersteps, the model provides a very straightforward approach to cost estimation by firstly, calculating the cost of each superstep, and secondly, calculating the cost of the whole BSP program by summing the cost of the supersteps. The cost of each superstep in a BSP program is given by:

$$T_{superstep} = w + hg + l$$

where w reflects the cost of the longest running local computation in any of the processors, l is a constant cost (the cost of the barrier synchronisation) that depends on the performance of the underlying hardware, h is the number of messages sent or received per processor and g captures the measurement of the ability of the communication network to deliver these messages. A number of parallel implementations have been proposed using the BSP model; see, for example, [125, 126, 127, 128].

A number of variants of BSP have been proposed. We briefly describe some here.

D-BSP: The Decomposable-BSP model[129] is a variant that allows submachine synchronisation. In the D-BSP model, the processors are grouped into several sub-machines that can synchronised independently, and each submachine M_i implements the BSP model using its own parameters(p_i, l_i, g_i).

E-BSP: The *Extended* BSP or E-BSP model [130] deals with locality and unbalanced communication. The E-BSP model targets communication patterns where the amount of data sent and received by each processor is different. Thus, E-BSP views each communication superstep as an (M,k1,k2)-relation instead of h-relation, where each processor sends at most k1 messages, receives at most k2 messages and the total of messages being routed does not exceed M.

MultiBSP: A recent extension to the BSP model is the MultiBSP model[131] for computation on modern architectures, in particular multicore architectures with memory/cache hierarchies. It uses four parameters to capture the characteristics of a multicore machine, namely, processor numbers, memory/cache sizes, communication cost, and synchronisation cost. MultiBSP is a hierarchical model that is based on a tree structure of nested components with an arbitrary number of levels. At each level, the model's parameters are assessed to allow all processors to execute independently until they reach barrier, and than they can all exchange information with the memory of that level.

4.1.3 The LogP Model Family

LogP[132, 133] is an architecture-independent parallel computation model for designing and analysing parallel algorithms. It is a model for distributed-memory multiprocessors where processors communicate using message passing. LogP provides a good balance between abstraction and simplicity by using a few parameters to characterise the parallel computers and enabling the user to ignore all unnecessary details.

Like the BSP model, LogP is more realistic, since both models try to capture the communication latency and bandwidth through parameters [134], and also both models allow the processors to work in a completely asynchronous manner. Nevertheless, the LogP model gives a more realistic picture than BSP, since LogP

has more control over the machine resources by capturing the communication overhead. Furthermore, LogP can be used in parallel systems that are constructed from a collection of complete computers connected by a communication network.

Conceptually, the LogP model consists of a collection of sequential processors interacting through a communication network by exchanging messages, where each processor has direct access to a local memory. The parallel program is executed in an asynchronous way by all processors in the LogP machine.

The LogP model seeks to capture the communication network cost by describing the parallel computer in terms of four elements:

P: the machine's number of **processors**.

g: communication bandwidth for short message (**gap**).

L: communication delay (**latency**) .

o: communication overhead (**overhead**) .

The latency is an upper bound on the time required to send a message from a source processor to its target processor. The overhead is the fixed amount of time that a processor requires to prepare for sending or receiving a message; during this time the processor cannot perform other operations. The gap is the minimum time interval between sending two messages on the same processor. The gap is the inverse of the available per-processor communication bandwidth for a short message. Several researchers[134, 135, 136] have shown that the LogP model delivers good and accurate predictions for small messages.

A number of different extensions of the classic LogP model have been developed to improve prediction accuracy by addressing different communication network issues:

LogGP

The LogGP model by Alexandrov et al.[137, 138] is an extension of the basic LogP model. Since LogP facilitates only short-message communication transmission between processors and ignores long messages, the LogGP model extended LogP to provides a simple linear model that can model both short- and long-messages.

Just as in the original LogP model, LogGP is developed for distributed-memory multiprocessors, where each processor has access to local memory. The processors work in an asynchronous way and communicate with other processors by point-to-point messages.

LogGP uses the parameters (*latency, overhead, gap, and number of processors*) that were introduced by the LogP model to characterise communication performance. In addition, it introduces a new additional parameter, Gap per byte, G, which captures the communication bandwidth for long message. Thus, the LogGP model uses *1/g* for short message and *1/G* for long message.

In the LogP model, sending a k bytes message between two processors requires sending $\lceil k/w \rceil$ messages, where w is the underlying message size of the machine. This takes:

$$o + (\lceil k/w \rceil - 1) * max(g, o) + L + o \quad cycles$$

while sending everything as a single large message in the LogGP model takes:

$$o + (k - 1) * G + L + o \quad cycles.$$

LogGPS

The LogGPS model[139] is a parallel computational model that extends LogGP to include the synchronisation cost. As in the original LogP model, LogGP eliminates the synchronisation cost that is needed in other models such as PRAM and BSP. This elimination might make LogGP not accurate enough, while it ignores the need for synchronisation when sending a long message in programs that use high-level communication libraries such as MPI. The LogGPS model has been proposed to address this shortcoming in LogGP.

Sending a long message between two processors is often performed by sending a small message to the receiver to check if it is ready to receive the original message. The process causes the sender processor to be synchronised with the receiver processor and adds a synchronisation cost to the overhead. Thus, the LogGPS model adds one additional parameter, **S**, which reflects the message-size threshold for synchronising sends.

HLogGP

Another extension of the LogGP model is the Heterogeneous LogGP[140] model. HLogGP has been specifically proposed for heterogeneous parallel systems to capture the heterogeneity in both communication networks and computational nodes. Since the underlying architecture of the LogGP model is very similar to the cluster architecture, it is considered an appropriate starting point for developing HLogGP.

The HLogGP model extends LogGP by transforming its scalar parameters into matrices. Conceptually, the parameters for overhead and gap are replaced by vector parameters, and latency and Gap is replaced by matrix parameters. Furthermore to capture the heterogeneity in the computational nodes, the parameter for the number of processors is replaced by a computational power vector, which describes the physical features for every node in the system.

The model has been shown to deliver an accurate prediction on heterogeneous clusters.

Other LogP extension

Besides the previously mentioned LogP extensions, other extensions have been proposed that aim to address different issues in communication. We briefly describe some here.

LogP-HMM[141] is a parallel computational model based on the LogP model.

The idea of LogP-HMM is to develop an accurate model that accounts for the impact of both network communication and multilevel memory on the performance of parallel algorithms and applications. Therefore, the LogP-HMM model extends LogP with the HMM model[142], where the LogP model deals with network communication and the HMM model addresses the memory hierarchy.

LoGPC[143] is a simple model that extends LogP and its extension LogGP to address another aspect of communication networks. It uses the features of both models to account for short message as well as long message bandwidth. Practically, the LoGPC model is intended to capture the impact of network contention on the performance of message passing programs. In addition, this model allows users to trade off between computation, communication and contention when designing parallel programs.

***parametrised* LogP**[144] or (**pLogP** for short) is a slight extension of the LogP and LogGP models. This model can accurately predict the completion time of collective operations in message passing models such as MPI. Five parameters are used in the pLogP model to characterised the network. Like the LogP model, it uses **P** as the number of processors and **L** is the end-to-end latency, but the original parameters **o** and **g** are replaced by a function of message size, where $o_s(m)$ and $o_r(m)$ are the sender and receiver overheads of the message size m, and *g(m)* is the delay between consecutive

message transmissions of size m.

4.1.4 HiHCoHP

The HiHCoHP model[145, 146] is a realistic communication model for hyperclusters (multi-level clusters of clusters of processors) with heterogeneous processors. It aims to capture the important features of a real hypercluster such as bandwidth and transmission cost.

The HiHCoHP model is based on several parameters that reflect the heterogeneity of hyperclusters:

- P_i *("computing power"):* HiHCoHP considers the computing power as N heterogeneous nodes that may differ in computational power (computation and memory speed).

- *(" message processing"):* the P_a and P_b set up fixed communication cost is $(\sigma_a^{(k)} + \bar{\sigma}_b^{(k)})$; and the cost of message packing in P_a is $\pi_a^{(k)}$ and message unpacking in P_b is $\bar{\pi}_b^{(k)}$.

- $\lambda^{(k)}$ *("network latency"):* or the end-to-end latency is the amount that is required to send one packet between the source node and destination node at level-k of the network.

- $\beta^{(k)}$ *("link-bandwidth"):* the amount of data that can be be sent between two nodes at level-k of the network.

- $k^{(k)}$ *("Network capacity"):* the maximum number of packets that can be transmitted at once.

So the total end-to-end communication time of sending p-packet message from node P_a to node P_b is given by:

$$(\sigma_a^{(k)} + \bar{\sigma}_b^{(k)}) + (\pi_a^{(k)} + \bar{\pi}_b^{(k)})p + \lambda^{(k)} + \Delta(p)$$

where $\Delta(p) = (p-1)/\beta^{(k)}$ in a pipeline network, and $\Delta(p) = \lambda^{(k)}(p-1)$ in a store-and-forward network.

4.1.5 DRUM

Another type of parallel computational model are architectures-aware cost models. One of the well-known models is the Dynamic Resource Utilisation Model[147, 148] or (DRUM). DRUM is developed to support resource-aware load balancing in a heterogeneous environment such as clusters and hierarchical clusters (clusters of clusters, or clusters of multiprocessors).

DRUM accounts for the capabilities of both network and computing resources. In particular, DRUM is intended to encapsulate information about the underlying hardware, and provide monitoring facilities for hardware capabilities evaluation. Benchmarks are used to assess the capabilities of computational, memory and communication resources.

Each node in the tree structure of the DRUM model has been given a single value called *"power"*, which represents the portion size of the total load that can be assigned to that node based on its processing and communication power. The power of node n in the DRUM model is calculated as the weighted sum of processing power p_n and communication power c_n:

$$power_n = w_n^{comm} c_n + w_n^{cpu} p_n, \quad w_n^{comm} + w_n^{cpu} = 1$$

4.1.6 Skeletons

To improve the performance of parallel applications, performance cost models are associated with algorithmic skeletons to accurately predict the costs of parallel applications. More precisely, the aim of these performance models is to assist the parallel skeletons, either implicitly or explicitly, to guide scheduling on a wide variety of architectures.

This section deals with skeleton-associated performance cost models. Several skeleton-based and similarly structured frameworks have employed performance cost models for various kind of skeletons. Some of the skeleton-based frameworks employ the well-known cost models and their variants such as the models that were previously mentioned, and others use their own performance prediction tools to estimate the performance of a given program.

Here, we briefly outline the skeleton-based frameworks that employ high-level cost models.

4.1.6.1 Darlington's group

Performance models are proposed in [149] for processor farms, divide and conquer (DC), and pipeline skeletons. For example, a performance model has been proposed for a divide and conquer skeleton to provide a prediction of the execution time for given program, which is used to guide resource allocation.

In this model, the total execution time required to solve a problem of size N on P processors is given by:

$$T_{sol_N} = \sum_{i=1}^{log(P)} (T_{div_{N/2i-1}} + T_{comb_{N/2i}} + T_{comms}) + T_{sol_{N/P}}$$

where T_{div_N} is the time to divide a problem of size N, T_{comb_N} is the time to combine the two results, and T_{comms} is the communication time between processors.

4.1.6.2 *BSP*-based Approaches

Several authors associate the BSP model with algorithmic skeletons for performance optimisation.

For example, Skel-BSP [126, 150] is a subset of P3L that uses an extension of the BSP model called the Edinburgh-Decomposable-BSP model to achieve performance portability for skeletal programming. EdD-BSP extends the BSP model by adding *partition* and *join* operations to partition and reunify BSP submachines which allows subset synchronisation as in D-BSP.

Compared to the standard BSP model, EdD-BSP replaces the g parameter with two parameters which are $g\infty$ and $N_1/2$, and then estimates the cost of two kinds of supersteps:

a) The cost of computational supersteps is given by:

$$T = W + hg\infty(N_{1/2}/h + 1) + L$$

b) The cost of partition and join superstep is given by L

Another *BSP*-based approach is Bulk-Synchronous Parallel ML (BSML)[151]. BSBML is a functional data parallel language for programming BSP algorithms using a set of high-level parallel primitives. It uses the BSP model to predict the performance of a given program on a wide variety of parallel architectures.

4.1.6.3 P3L

P3L uses a variant of the LogP model to predict and optimise program performance on parallel systems. An analytic model is presented in [152] for the basic forms of parallelism to be used by the template-based compiler of the P3L language.

This model is more complex than LogP, since it is intended to capture several hardware features, such as the speed of processor, node architecture, and network

bandwidth and latency.

Here we briefly describe the analytical model for the high level template that is related to our work.

The **Map** construct is implemented on an N dimension grid of processors. The computation time T of input granularity k is given by:

$$T(k) = k(T_{dis}() + T_c \prod_{i=1}^{N} d_i + T_{col})$$

where:

T_c : seq. computation time.

d_i : data granularity for dimension i.

T_{dis}: data distribution time.

T_col: time for collecting results.

4.1.6.4 HOPP

The HOPP (Higher-order Parallel Programming) model[153, 154] is a methodology based on the BMF (Bird-Meertens Formalism)[155], where the program is expressed as a composition of higher-order functions.

The HOPP model uses a cost model introduced in [156] to predict the costs of programs. This cost model is implemented as an analyser for calculating the costs of possible implementations for a given program on a given distributed-memory machine.

In the HOPP model, the cost of a program is computed in terms of n steps:

$$Cost = \sum_{i=1}^{i=n} C_{p_i} + \sum_{i=0}^{i=n-1} C_{i,i+1}$$

where C_{p_i} is the cost of phase i which depends on the number of processors and sequential implementation of the functions in that step, and $C_{i,i+1}$ is the cost of communication that may be incurred between step i and step $i+1$.

4.1.6.5 SkelML

SkelML[157] gives performance models for a number of skeletons such as pipeline, farm, and fold Processor Chain skeletons. These models are based on the communication overhead and computation time that are involved in application execution. The skeleton performance models and profiling information help the SkelML compiler to determine useful parallelism.

4.2 Resource Metrics for Parallel Cost Models

The performance of parallel machines is dependent on the underlying architecture features. These features are referred to as resource metrics that characterise the parallel computational model. Thus, a computational model can be identified by a set of these resource metrics. We now consider some resource metrics that are

visible in all parallel computational models that were discussed in Sections 4.1.

Number of processors. The number of processor in the machine.

Communication Latency is the time needed to transfer a message from one processor to another processor; this depends on both the network topology and technology.

Communication Bandwidth is the amount of data that can be sent within a given time; this is a limited resource in practice and depends on the network interface.

Communication Overhead is the period of time that is needed by the processor for sending and receiving message. The amount of overhead depends on network topology features such as communication protocols.

Computational power. Computational power is the amount of work finished by one processor in a given time for a specific task; this value depends on the processor's capabilities and the task being processed.

Synchronous/Asynchronous. In a synchronous model, all processors are synchronised after executing each instruction. Processors may run semi-asynchronously, where the computations occur asynchronously within each phase and all processors are synchronised at the of each phase.

Table 2 shows how these resource metrics contribute in forming the computational models considered above.

Model	Procs	Latency	Bandwidth	Overhead	Computational Power	Synch Asynch
PRAM	√					Synch
BSP	√	√	√			Semi-synch
LogP	√	√	√	√		Asynch
LogGP	√	√	√	√		Asynch
HLogGP	√	√	√	√	√	Asynch
SkelML	√			√		Asynch
P3L	√	√	√	√	√	Asynch
Ske-BSP	√	√	√			Semi-synch
HOPP	√	√	√	√	√	Asynch

Table 2: Resource metrics for parallel computation.

4.3 Design Ethos

In common with other cost-based skeletal approaches, our approach combines algorithm skeletons with a performance cost model that characterises a parallel machine, using performance parameters.

The current focus of designing parallel performance cost models is on providing low-level details of parallel execution to the programs to enable resource-aware partitioning and dynamic load balancing procedures, in particular, for heterogeneous parallel architectures. We claim our methodology presented in Section 4.4 based on architectural details of a parallel machine to provide cost estimation of a given program on a given machine, provides a reasonable trade-off between the accuracy and simplicity needed for our heterogeneous skeletons.

Static cost models incur less overhead than dynamic models due to their simplicity and lack of run-time overhead by eliminating the most costly dynamic parameters such as network parameters. Thus, we introduce an architecture-aware static cost model that accounts for a few simple architectural parameters which reflect the processing capabilities that affect load-balancing on heterogeneous architectures.

In contrast to the cost models described in Section 4.1, our cost model provides the following features:

Relative Simplicity. As the degree of model complexity depends on the numbers of parameters that need to be estimated, we are content with a simple model, with small a number of parameters.

Target Architectures. HiHCoHP and HLogGP models are designed for heterogeneous clusters in which both processors and network are heterogeneous. Nevertheless, our cost model targets homogeneous networked cluster where the nodes are heterogeneous. We focus on the optimisation of processing time that can be affected by the computational features of the nodes.

Skeleton-based Approach. The idea of associating cost models with algorithmic skeletons is not new. However, we are integrating an architectural cost model that accounts for cache size for load balance in heterogeneous clusters into a skeleton library for parallel implementations. Moreover, the cost

model is used implicitly to guide the implementation of parallel programs.

Performance Optimisation We seek to provide a simple cost model to optimise overall processing time for our skeletons on a heterogeneous system, rather than extracting the maximum performance from heterogeneous systems.

4.4 The CM1 Cost Model

We develop a cost model to optimise overall processing time for our skeletons on a heterogeneous system composed of networks of arbitrary numbers of nodes, each with an arbitrary number of cores sharing arbitrary amounts of memory and arbitrary clock speeds. From our model, we seek *relative* measures of processing power to guide data distribution rather than *absolute* predictions of processing time. Thus, we are content with a simple model, with small numbers of easily instantiable parameters.

In constructing the CM1 cost model, we assume that:

- inter-node communication time is uniform;
- on an individual node, all the cores have the same processor characteristics;
- each core processes a distinct single chunk of data, without interruption, using the same algorithm as the other cores;

Hence, we focus the optimisation of processing time on distributing appropriately sized chunks of data to cores to balance processing.

As a first attempt we might base this distribution on the number of nodes, and for, each node, the speed of each core. Suppose node i has C_i cores each of speed S_i. Then, the total available processing power for n nodes is:

$$\sum_{i=1}^{i=n} C_i * S_i$$

Each node i might receive:

$$C_i * S_i / \sum_{i=1}^{i=n} C_i * S_i$$

of the data, so each core of node i might receive:

$$S_i / \sum_{i=1}^{i=n} C_i * S_i$$

Now, all processors have some memory hierarchy, from registers, via various levels of cache, to RAM and beyond. We assume that registers and on-core caches are private and operate at CPU speed. Shared cache, typically L2 or L3, is usually many orders of magnitude smaller than shared RAM, and, for many problems, RAM is sufficiently large for paging to be absent. Thus, we identify the size of top level shared cache as the most significant memory factor affecting overall performance.

The SPMD model implies that all cores are running the same algorithm, which implies that they will have similar patterns of access to shared memory. In particular, each core will incur similar sequences of cache faults. Then, the number of cache faults will be determined by the size of the cache: for a larger cache it is more likely that a required portion of the address space is already resident.

Thus, we refine our model to take into account the size of the cache, which we denote as $L2_i$ on node i, with a larger cache implying that a node should receive larger size data chunks. Then, the relative power of node i of heterogeneous multicore cluster is given by:

$$C_i * S_i * L2_i \tag{7}$$

The overall power P of the system is given by:

$$P = \sum_{i=1}^{i=n} C_i * S_i * L2_i \tag{8}$$

For total data size D, the chunk size for node i is:

$$(C_i * S_i * L2_i/P) * D \tag{9}$$

and each core processes:

$$(S_i * L2_i/P) * D \tag{10}$$

For a heterogeneous multicore system, it is necessary to normalise the overall system power in order to predict maximum speedup and determine whether that has been achieved as will be discussed in section 4.6.2.3. We think it most principled to do so using *the core with the greatest power*.

So, to predict maximum speedup we:

- find the power of each core P_i and choose the greatest P_l:
- find the maximum possible speedup by dividing the overall power by the greatest core power: P/P_l.

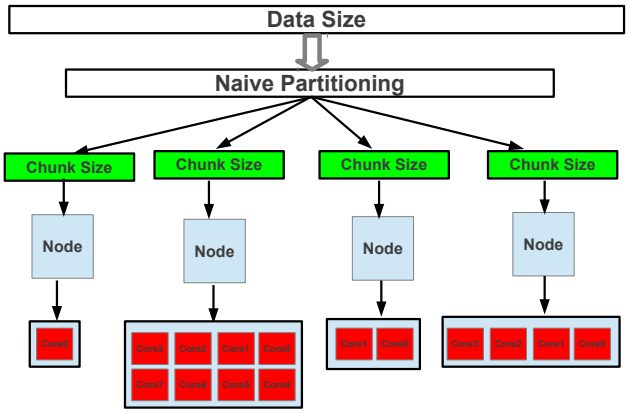
Then, to assess achieved speedup we:

- initially, measure the program on one core with that greatest power to provide a base line;
- subsequently, measure speedup relative to that base line measurement.

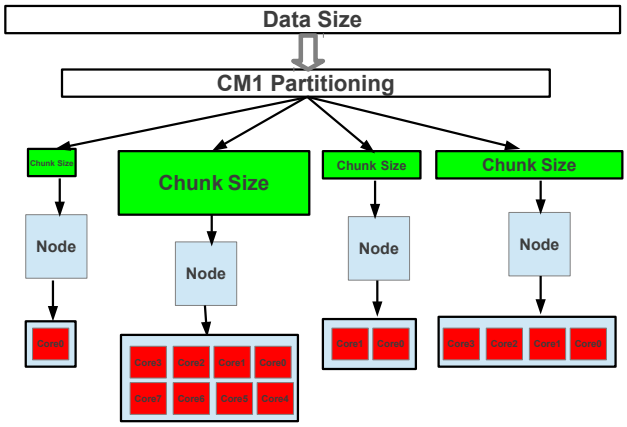
4.5 Using CM1 in the *HWSkel* Library

In the *HWSkel* library the CM1 cost model is integrated into the skeletons to improve their parallel performance on heterogeneous multicore clusters. In the hybrid programming model, load balance can be more easily achieved in the shared-memory model(OpenMP) than the distributed-memory model(MPI), hence the load balance is dependent on MPI distribution not on OpenMP.

Since the implementation of all skeletons in the *HWSkel* library is based on a hybrid programming model where we assume that all cores on the node have the same characteristics, we use CM1 only for data distribution and hence load over the cluster nodes. Figure 8 (a) illustrates a naive load balancing mechanism for load distribution, and (b) shows the effects of the hardware-based cost model on load balancing for data distribution in *HWSkel*.



(a)



(b)

Figure 8: Using Cost Model (CM1) in *HWSkel* for Load Distribution

Our data-parallel heterogeneous skeletons use the SPMD model for distributed memory parallelism, therefore the master Processing Element (PE) is responsible for applying CM1 as discussed in Section 3.2. Every skeleton program initially calls *InitHWSkel* that, collects and registers the architectural information for each node in the cluster that is needed by CM1 as discussed in Section 3.2.2.1. This information is collected from the local system file "/proc/cpuInfo" of each node in the system. After collecting the hardware information, the master node applies CM1 to distribute the data over the cluster. The algorithm for implementing CM1 in our skeletons is displayed in Algorithm 4, and the complete code for the CM1 cost model can be found in Appendix B.1.

Algorithm 4 The Implementation of The CM1 Cost Model

1: ***skeleton initialisation***
2: BEGIN
3: → *master node:*
4: **for** every node$_i$ **do in parallel**
5: list[i] ⇐ node_SPECs ($C_i * S_i * L2_i$) /* hardware specifications */
6: **end for**
7: **for** every list[i] **do**
8: $P+ = C_i * S_i * L2_i$ /* overall power of the system */
9: **end for**
10: **for** every list[i] **do**
11: chunksList[i] ⇐ $(C_i * S_i * L2_i/P) * D$ /* calculate chunk size */
12: **end for**
13: **for** every node$_i$ **do in parallel**
14: Send chunklist[i]
15: **end for**
16: END

4.6 *HWSkel* Evaluation

This central section of the thesis details and explains the experiments that were performed to provide evidence of the effectiveness of our cost model CM1 for improving the performance of programs using *HWSkel* skeletons. In addition, we illustrate and demonstrate the validity of our cost model by comparing the the results obtained with those from several alternative related cost models that use different architectural parameters.

4.6.1 Benchmarks

We have assessed the impact of our cost model on *HWSkel* heterogeneous skeletons using two different applications.

4.6.1.1 *sum-Euler*

The *sum-Euler* benchmark calculates the sum of the Euler totients between a lower and an upper limit, where the totient function of an integer n gives the number of positive integers less than or equal to n that are relatively prime to n:

$$sumEuler = \sum_{n=lower}^{upper} \phi(n)$$

$$\phi(n) = \sum_{i=1}^{n} euler(i)$$

Code fragment 4.1 is a sequential `sumTotient` function that receives a list of integers.

```
1
2  int sumTotient(int *datalist, int length)
3 {
4    int i,j,k;
5    int sum;
6    sum = 0;
7    for(i=0;i<length;i++)
8    {
9      sum = sum+euler(datalist[i]);
10   }
11   return sum;
12 }
```

Listing 4.1: Code for `sumTotient` function.

Code fragment 4.2 shows the `euler` function that applies the Euler totient function to each element in the array; then the results are summed for all elements. In the parallel version, the array of the integers is split into chunks using a split function which employs the cost model of load distribution, and then the `euler` function is mapped in parallel across each chunk.

```
1
2  int euler(int n)
3 {
4    int i;
5    int length=0;
6    for(i=1;i<n;i++)
7    {
8       if(relprime(n,i)){
9           length++;
10      }
11   }
12   return length;
13 }
```

Listing 4.2: Code for `euler` function.

Finally, the results are summed sequentially for all elements in the main function(`sumTotient`). Code fragment 4.3 presents the main *sum-Euler* program that uses the *hMapReduce* skeleton.

```
1
2  int main(int argc, char **argv)
3  {
4      InitHWSkel(argc,argv);
5
6      result=hMapReduce(data,length,INT
7                        ,sumTotient,INT,plus);
8
9      TerminateHWSkel();
10 }
```

Listing 4.3: Main program for *sum-Euler*.

The parameters of *hMapReduce* skeleton are `sumTotient` as the map function and the plus from Program 4.4 as the reduction function.

```
1
2  int plus(int *arr,int size)
3  {
4      int i;
5      int result=0;
6      for(i=0;i<size;i++)
7      {
8         sum+=arr[i];
9      }
10     return result;
11 }
```

Listing 4.4: Code for `plus` function.

4.6.1.2 Image Matching

Image Matching is a fundamental aspect of many problems in computer vision including object recognition. Matching different images of an object requires local image features that are unaffected by nearby clutter or partial occlusion [158]. The Scale Invariant Feature Transform (SIFT) is an approach used to transform image data into scale invariant coordinates relative to local features which has properties that make it suitable for image matching and recognition [159]. Therefore, image matching is performed by first extracting local features from the input image using a SIFT algorithm and then these features are individually matched

to SIFT features obtained from training images by using a nearest-neighbour algorithm. In addition, to avoid the expensive search required for the nearest-neighbour algorithm, a modification of the k-d tree algorithm called best-bin-first method is used [158].

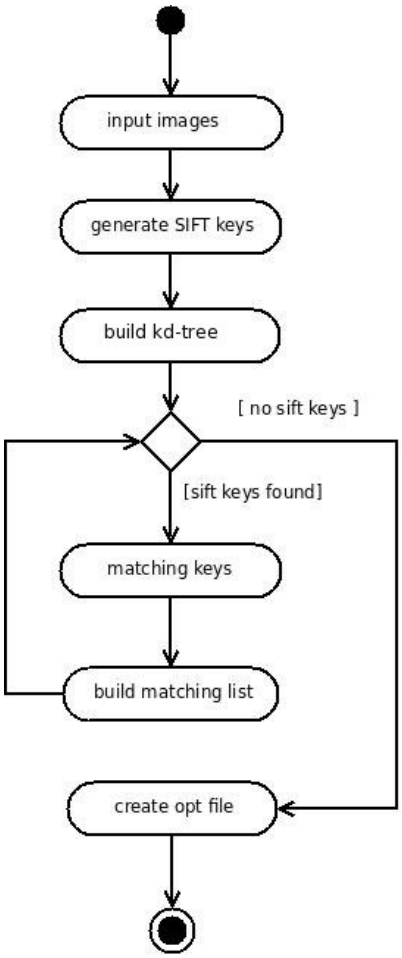


Figure 9: Flowchart of Sequential Image Matching Algorithm

We ported the sequential object recognition program written by David Lowe at the University of British Columbia. This application consist of 26 files which contain approximately 9500 lines of C code. Basically, the sequential algorithm

divides into two stages: the first stage is SIFT keypoints detection and secondly there is SIFT keypoints matching. The flowchart in Figure 9 illustrates the original sequential algorithm.

The parallel version parallelises the computationally expensive second stage of the application using the *hMapReduceAll* skeleton. That is, the application is parallelised by allocating the keypoints to the cores and each core applies the BBF algorithm [159] in order to perform the matching operation.

Code fragment D.5 presents the main *Image Matching* program using the *hMapReduceAll* skeleton where the *matchKeys* is the map function and the reduction function is *gatheringKeys*.

```
1
2  #include"MatchBoth.h"
3
4  #include"HwSkel.h"
5
6  int main(int argc, char **argv)
7  {
8     ArrayList* keys;
9     int keysCount;
10
11    //- skeleton initialization
12    InitHWSkel(argc,argv);
13
14    //- allocate memory for an arraylist of keypointNs
15    MultiMatch* mm = MultiMatch_new0();
16
17    if(StartNode)
18    {
19       int no;
20
21     //- number of input images
22       mm->imageCount = 2;
23       mm->keySets=ArrayList_new(mm->imageCount,KeypointXMLList_delete);
24
25       for(no=2;no<4;no++)
26       {
27          //- 1. load the image file
28          DisplayImage* pic = DisplayImage_new(argv[no]);
29          ImageMap* picMap = DisplayImage_ConvertToImageMap(pic);
30
31          //- 2. find the image features (keypointNs)
32          mm = getImageFeatures(picMap);
33       }
34
35       keysCount=ArrayList_Count(mm->globalKeys);
```

```
36          keys = keyN_MPI_Pack(mm->globalKeys);
37      }
38
39      //- call hMapReduceAll skeleton
40      ArrayList* list = hMapReduceAll(keys,keysCount,ARRAY_LIST,matchKeys,
            ARRAY_LIST,gatheringKeys);
41
42      if(StartNode)
43          printf(" Global match search yielded: %d  Match \n",ArrayList_Count(list))
                ;
44
45      //- skeleton termination
46      TerminateHwSkel();
47
48      return 0;
49  }
```

Listing 4.5: Main program for image matching.

4.6.2 Performance Evaluation

In this experiment we investigate the performance impact of both *hMapReduce* and *hMapReduceAll* skeletons on homogeneous shared memory architectures, and on different combinations of heterogeneous multicore architectures using CM1 for load distribution.

Moreover, we use this experiment to study the contribution of each hardware property that is used in the cost model. In our experiments, we have matched two input images, where the size and the number of SIFT keypoints for each image are shown in Table 3.

Since the original sequential sum-Euler program generates irregular data granularity, for simplicity we assume that all the elements in the array have the same value and calculate the sum of the totients between 1 and 2,000,000 of an integer with fixed value of 10,000 *i.e.* [10000, 10000, ... ,10000].

	Size	**Keys**
img1	1600x1200 (239,616)	71791
img2	1600x1200 (1,205,862)	12378

Table 3: Input Images for Image Matching Application.

4.6.2.1 Platform

We conduct our experiments on a heterogeneous cluster of five different parallel architectures as summarised in Table 4. The employed machines are located at Heriot-Watt University

- ***linux_lab:*** 2-core machines consisting of Linux RedHat 4.1.2 workstations with a 2.4GHz Intel processor, using 2GB RAM and 2048KB L2 cache.

- ***lxpara:*** 8-core Dell PowerEdge 2950 machines constructed from two quad-core Intel Xeon 5410 processors running Linux RedHat 5.5 at 2.3GHz with 6144 KB L2 cache and using 8GB RAM.

- ***amaterasu:*** a 4-core machine running Linux RedHat 4.1.2 at 2.93GHz with 8192 KB L2 cache and using 16GB RAM.

- ***brahma:*** a 4-core machine running Linux RedHat 4.1.2 at 3.06GHz with 512 KB L2 cache and using 4GB RAM.

- ***jove:*** a 8-core machine running Linux RedHat 4.1.2 at 2.80GHz with 8192 KB L2 cache and using 16GB RAM.

Throughout the evaluation section, the architectures will be cited as *(speed/cache)* e.g (3G/512KB).

Three experimental combinations were explored (Het.Arch1-3). In each combination, new machines were successively added from strongest to weakest.

Further, Compilation of benchmarks has been done using GCC 4.1.2 with the *-fopenmp* flag for OpenMP compilation. Since parallel runtimes are variable, the measurements are based on the middle (median) value of three executions.

Machine name	**Architecture**	**Cores**	**MHz**	**L2 Cache**	**Het. Arch1**	**Het. Arch2**	**Het. Arch3**
lxpara1	Xeon 5410	8	1998	6144KB	√	√	√
lxpara2	Xeon 5410	8	1998	6144KB		√	
brahma	Xeon(TM)	4	3065	512KB	√	√	√
amaterasu	Core(TM) i7	4	1199	8192KB	√	√	√
jove	Core(TM)i7	8	1200	8192KB	√		
linux01	2 Duo CPU	2	1200	2048KB	√	√	√
linux02	2 Duo CPU	2	1200	2048KB	√	√	√
linux03	2 Duo CPU	2	1200	2048KB			√
linux04	2 Duo CPU	2	1200	2048KB			√

Table 4: Experimental Architectures.

4.6.2.2 Homogeneous Architectures

On homogeneous architectures both skeletons deliver linear speedup for the *sum-Euler* and *Image Matching* programs.

Figure 10 compares the speedup curves for *sum-Euler* with an OpenMP version and hMapReduce skeleton in a shared-memory environment (*i.e. lxpara*).

Both speedup curves are similar and hence hMapReduce is efficient on shared-memory architectures.

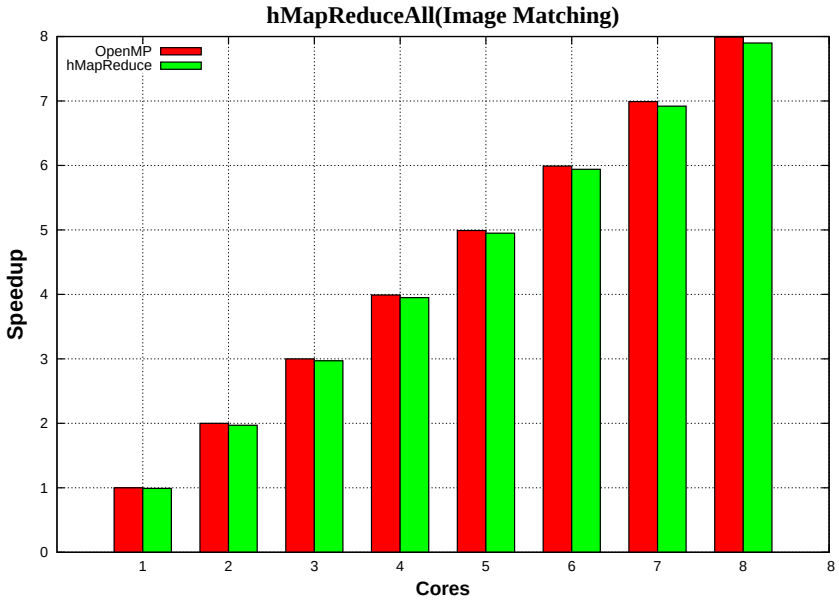
4.6.2.3 Heterogeneous Architectures

On heterogeneous architectures, we run our skeletons on three different combinations of the architectures described in Section 4.6.2.1.

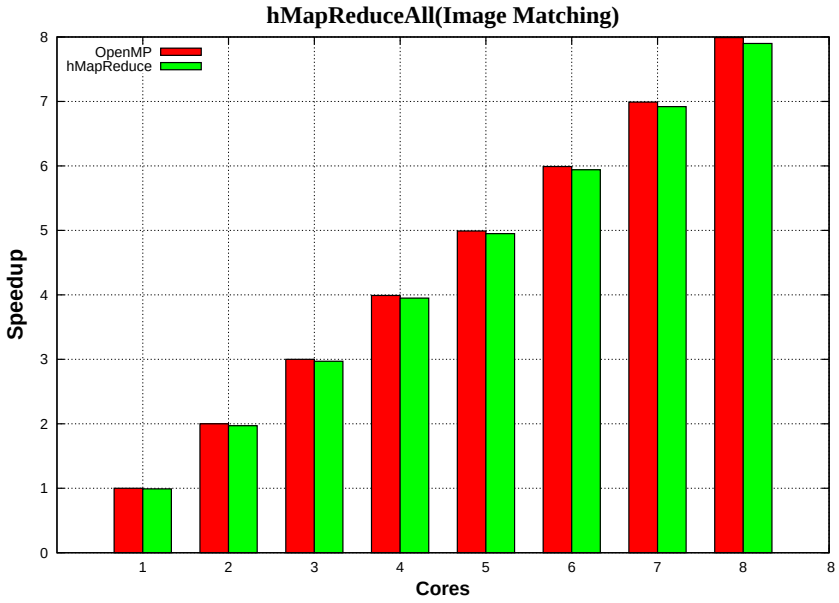
Heterogeneous Architecture 1. Figure 11 plots speedup curves for our testbed parallel programs on Het.Arch1 (*lxpara*, *brahma*, *amaterasu*, *jove* and 2x*linux*). Observe that the predicted and experimental speedup curves for *Sum-Euler* are identical.

The results show that the implementation of *HWSkel* library without the cost model delivered worse scalability, where we achieved good speedup on the first fast machine (*lxpara*). The lower speedup curve falls as soon as we introduce heterogeneity by adding the slow machines. This is due to the naive load balancing mechanism which distributes load equally between the machines. The upper speedup curve shows the improved performance results for using a load distribution based on the cost model in Section 4.4. As anticipated, our results show better scalability for our skeletons with the CM1 cost model.

Heterogeneous Architecture 2. We combine two 8-core shared-memory machine *lxpara* with 4-core shared-memory machine *brahma*, 4-core shared-memory machine *amaterasu* and two 2-core shared-memory machine *linux* (Het.Arch2).

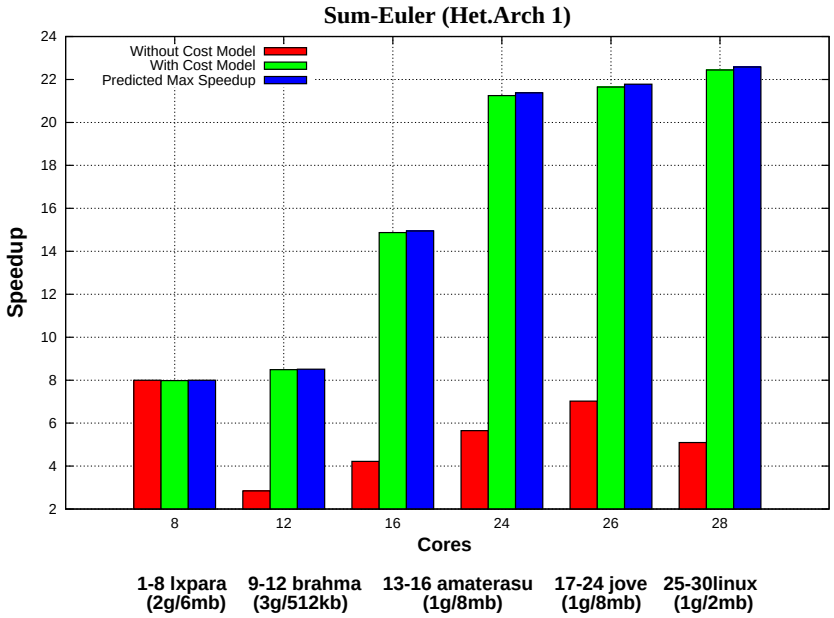


(a)

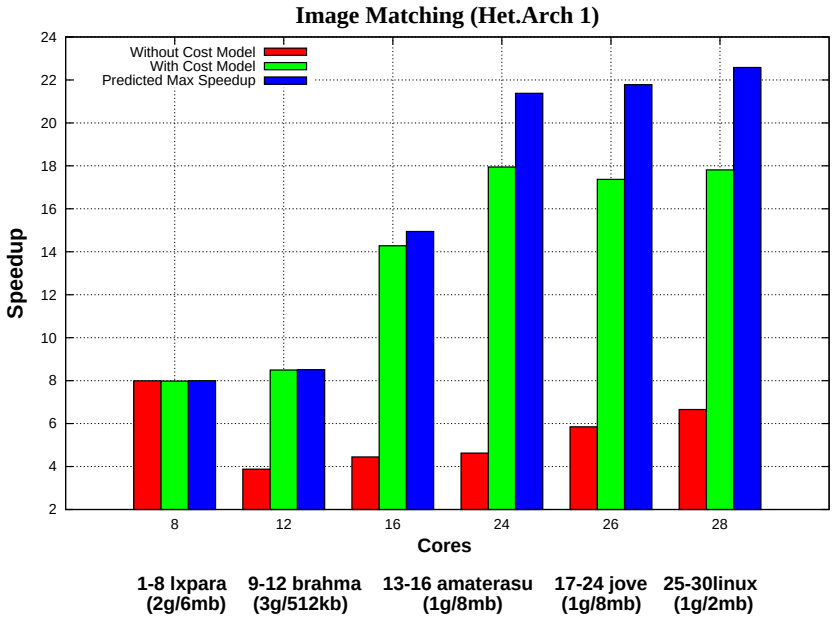


(b)

Figure 10: Comparing hMapReduce(sum-Euler) and hMapReduceAll(Image Matching) with OpenMP on Shared-Memory Architectures (lxpara).

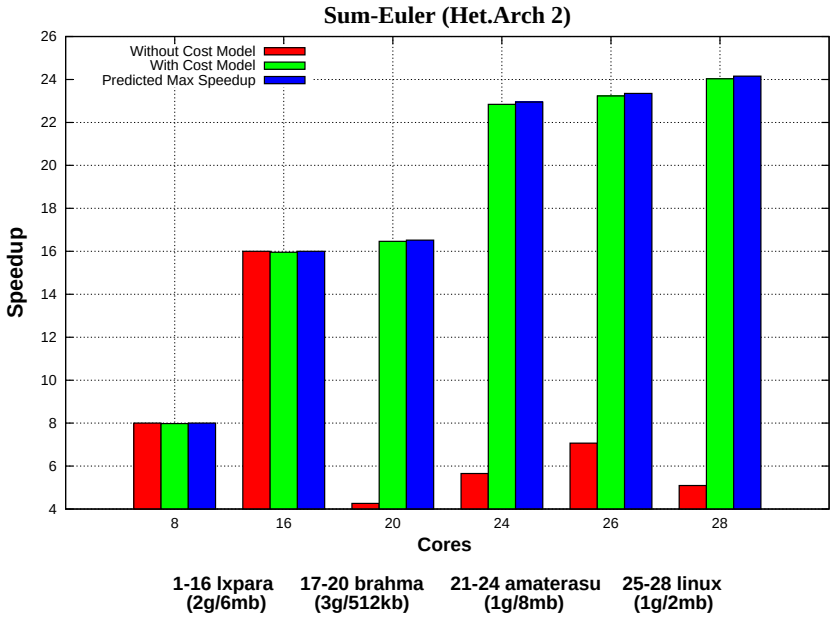


(a)

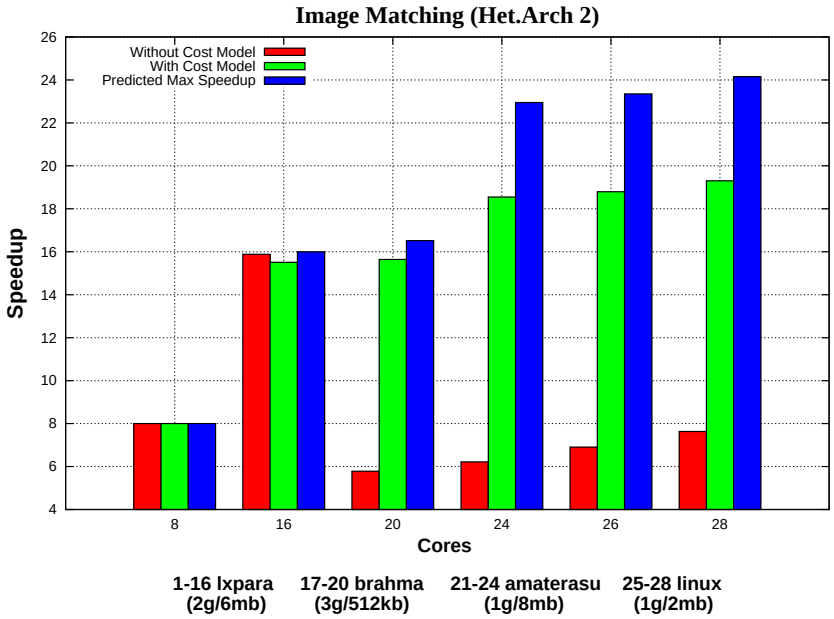


(b)

Figure 11: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with/without Cost Model on (Het.Arch1)



(a)



(b)

Figure 12: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with/without Cost Model on (Het.Arch2)

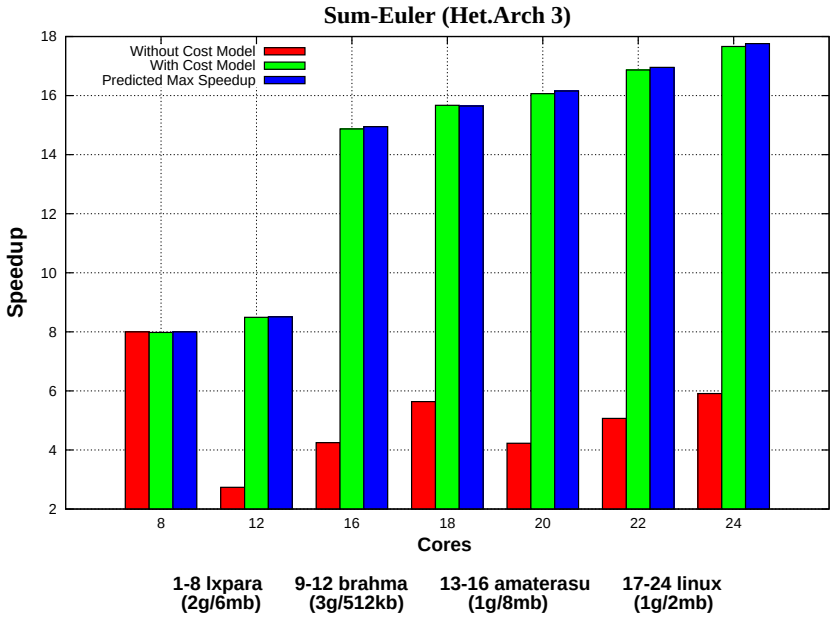
Observe that the predicted and experimental speedup curves for *Sum-Euler* are identical.

The results in Figure 12 show that in the first two machine (*lxpara1* and *lxpara2*) the performance of our implementations of both *hMapReduce* and *hMapReduceAll* skeletons are slightly improved by using the cost model. This is due to the architectural similarity of these machines. As for Het.Arch1 adding slow machines leads to poor performance due to the data-load distribution mechanism. Again Figure 12 shows the performance of our skeletons can be improved using the cost model.

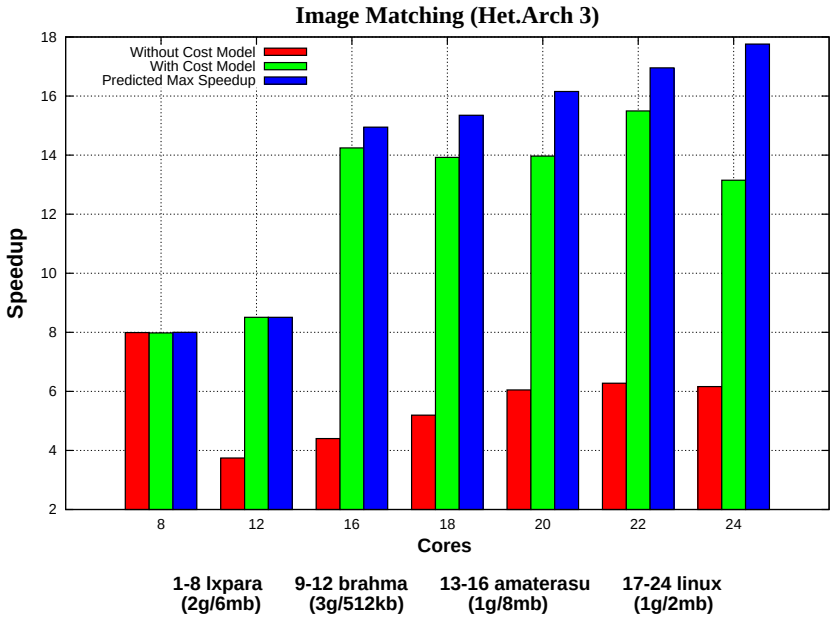
Heterogeneous Architecture 3. Figure 13 shows the speedups on a different heterogeneous architecture (Het.Arch3) comprising (*lxpara*, *brahma*, *amaterasu*, and 4x*linux*). Observe that the predicted and experimental speedup curves for *Sum-Euler* are identical.

For this combination, the results look similar to the first combination shown in Figure 11. It shows that the performance of our skeleton is improved by using the CM1 cost model.

Predicted Maximum Speedup. In order to assess the effectiveness and accuracy of the CM1 cost model for our heterogeneous skeletons, we calculate the predicted maximum speedup as described in section 4.4 and compare it with the experimental speedup for both programs. Tables 5, 6 and 7 lists the predicted speedup, experimental speedup and the relative error on the 3 heterogeneous



(a)



(b)

Figure 13: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with/without Cost Model on (Het.Arch3)

Predicted Speedup	**Experimental-Speedup**				**Relative Error**	
	Img-Match	**STD**	**Sum-Euler**	**STD**	**Img-Match**	**Sum-Euler**
8	7.99	0.0227	7.99	0	0.125 %	0.125 %
8.51	8.694	0.1481	8.492	0.0005	-2.162 %	0.211 %
14.946	14.28	0.0258	14.89	0.0005	4.456 %	0.374 %
21.38	17.949	0.2199	21.25	0.0005	16.047 %	0.608 %
21.782	17.371	0.0351	21.649	0.0008	20.250 %	0.610 %
22.58	17.815	0.0238	22.44	0	21.107 %	0.620 %

Table 5: Experimental and Predicted Maximum Speedup (Het.Arch1).

Predicted Speedup	**Experimental-Speedup**				**Relative Error**	
	Img-Match	**STD**	**Sum-Euler**	**STD**	**Img-Match**	**Sum-Euler**
8	8	0.0221	7.981	0	0 %	0.238 %
16	15.507	0.0593	15.955	0	3.081 %	0.281 %
16.522	15.64	0.0493	16.467	0.0005	5.338 %	0.332 %
22.95	18.549	0.1133	22.84	0	19.176 %	0.479 %
23.353	18.794	0.2188	23.236	0.0005	19.522 %	0.501 %
24.157	19.304	0.2097	24.033	0.0008	20.089 %	0.513 %

Table 6: Experimental and Predicted Maximum Speedup (Het.Arch2).

architectures, and also show the standard deviation of runtime over three runs.

In all 3 architectures, the relative error for *Sum-Euler* program is very low where the experimental speedup is close to the maximum theoretical speedup predicted by our cost model. However, as expected in the *Image-Matching* program the relative error is higher. The error is around 25.956 percent in the worst case. This is due to the characteristics of this program which suffers high overheads because of frequent communication. Figures 11, 12 and 13 plot the predicted maximum speedup for the *Sum-Euler* and *Image-Matching* programs.

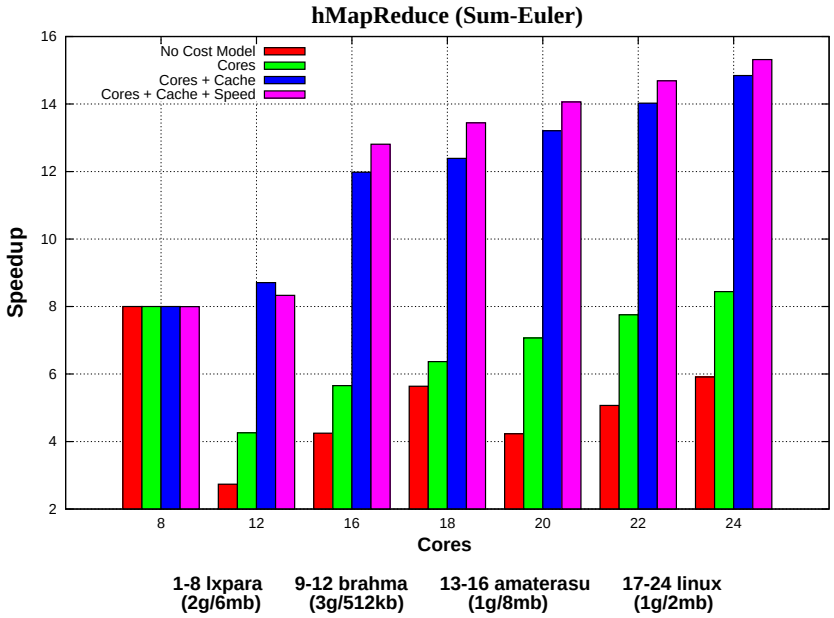
Predicted Speedup	Experimental-Speedup				Relative Error	
	Img-Match	STD	Sum-Euler	STD	Img-Match	Sum-Euler
8	8	0.0189	7.981	0.0005	0 %	0.237 %
8.51	8.694	0.0443	8.492	0	-2.162 %	0.211 %
14.946	14.244	0.0143	14.87	0.0008	4.696 %	0.508 %
15.348	13.923	0.4380	15.669	0.0005	9.284 %	-2.091 %
16.153	13.967	0.0253	16.067	0.0008	13.533 %	0.532 %
16.957	15.495	0.0167	16.864	0.0005	8.621 %	0.548 %
17.762	13.152	0.0184	17.661	0.0005	25.954 %	0.568 %

Table 7: Experimental and Predicted Maximum Speedup (Het.Arch3).

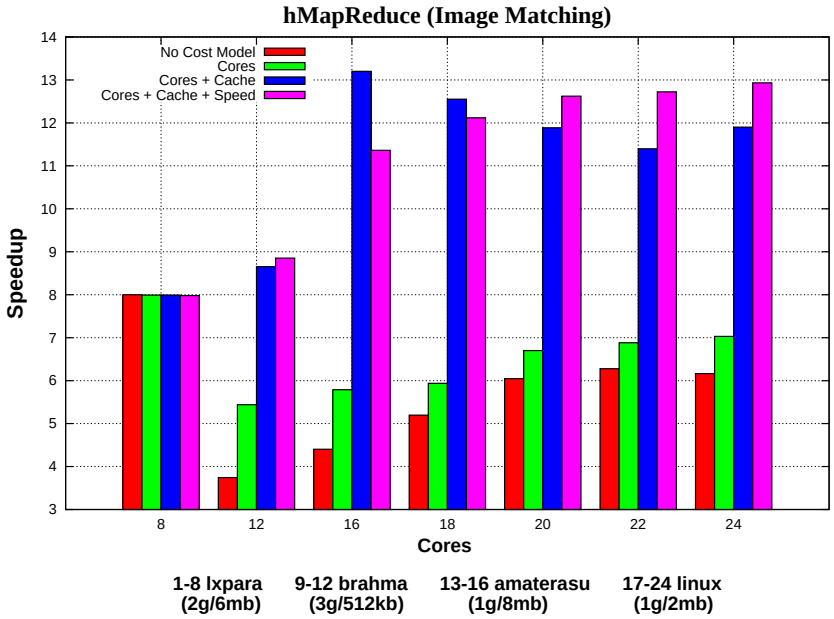
4.6.3 Alternative Cost Models

Figure 14 shows different speedup results for the implementations of *HWSkel* library on Het.Arch3 (*lxpara*, *brahma*, *amaterasu*, and 4x*linux*) using the CM1 cost model with different architecture properties which includes number of cores, CPU speed, and cache size. Although, the best result is achieved by using all CPU properties in the cost model, we can see that the cache size property has the most significant impact on the cost model performance.

Therefore, we conclude that our skeletons can deliver good parallel performance and scalability on heterogeneous architectures using the static load-balancing mechanism based on architecture properties. On the architectures that are likely to be more heterogeneous the communication cost needs to be added to the cost model.



(a)



(b)

Figure 14: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with Alternative Cost Models on (Het.Arch1)

4.7 Summary

In this chapter, we have proposed a new architectural performance cost model CM1 for heterogeneous parallel architectures. CM1 is used to statically determine the data-chunk size according to the number of cores, clock speed and crucially the cache size for each node over the heterogeneous multicore cluster

Since the naive implementation of our skeletons on a heterogeneous multicore cluster delivers poor performance due to the difference in node capability, we have shown in this chapter that it is possible to obtain better performance using CM1 for workload distribution by our data parallel heterogeneous skeletons on a heterogeneous multicore platform, which in turn makes the task of workload distribution much easier for the skeleton programmer.

Finally, our experiments have shown that the cache size has the most significant impact on the data-load distribution mechanism.

Chapter 5

GPU-HWSkel Library

This chapter describes our *GPU-HWSkel* skeleton library that extends the *HWSkel* library to account for GPU programming. We start by introducing *GPU-HWSkel* library in Section 5.1. Next We describe the general architecture of our system in Section, which implemented by heterogeneous programming model in Section 5.2. In Sections 5.3 and 5.4 we present the user functions and *GPU-HWSkel* skeletons.

5.1 *GPU-HWSkel: A CUDA-Based Skeleton Library*

GPU-HWSkel is an extension of the *HWSkel* library introduced in Chapter 3. The library is designed with the aim of providing a high-level parallel programming environment to program parallel heterogeneous multicore/GPU systems including single- and multicore CPU, and GPU architectures.

The *GPU-HWSkel* library is based on the CUDA programming model to make GPGPU accessible on NVIDIA GPUs. This limits our approach to NVIDIA architectures. However, *GPU-HWSkel* can potentially take advantage of the OpenCL standard to make other GPU devices accessible through GPU-HWSkel-*based* skeleton programming.

Like SkePU, *GPU-HWSkel* offers a user function that can be used as an argument to our heterogeneous parallel skeletons described in Section 5.4. However, our approach is implemented by using a different programming model, which we will discuss in the following section.

The new library implements the same set of data-parallel skeletons that are provided by the base *HWSkel* library, *i.e.* hMap, hReduce, hMapReduce, and hMapReduceAll. These skeletons provide a general interface for both GPUs and CPUs since the library is based on OpenMP and MPI to support CPU implementations, and CUDA for GPU implementations.

Theoretically, the library takes into account the support of Multi-GPU systems since most parallel systems provide multiple GPU cards to increase the number of processing units for high performance as discussed in Section 7.3.2.

5.1.1 *GPU-HWSkel* Implementation Principles

The key idea behind *GPU-HWSkel* is to generate CUDA code (kernel functions) at compilation time. After the compilation stage, all CUDA kernels are ready to be executed. These kernel functions are passed to the skeletons as parameters for

execution on the GPU.

Since the design aim of the *GPU-HWSkel* library is to provide developers with a set of heterogeneous skeletons which can be used on a variety of parallel platforms, both sequential and parallel code, for multicore and GPU architectures, are generated for multiple implementation support.

The selection between different implementations of the *GPU-HWSkel* library depends on the available architectures that are provided by the target hardware. Therefore, an automatic-implementations selection plan for different skeletal implementations is implemented in each of our skeletons. So as shown in Figure 15, the auto-selection plan works by executing suitable code from the generated codes for the available underlying hardware on either a sequential or parallel architecture, where Phase 1 is generating CPU & GPU code from the input program, and phase 2 is an auto-selection plan to select different skeletal implementations.

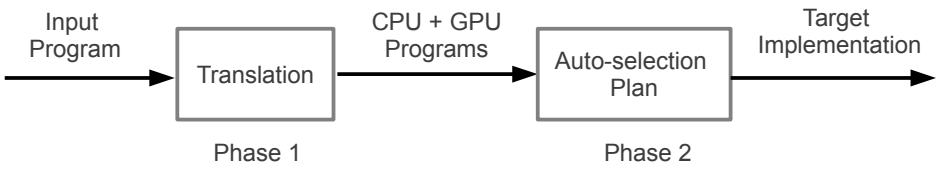


Figure 15: Automatic-Implementation Selection Plan.

In most cases of heterogeneous clusters, the underlying hardware of each node comprises both a multicore CPU and a GPU. Thus, the *GPU-HWSkel* library

is implemented to enable CPU cores and GPU to execute the same computational operations concurrently as follows: *GPU-HWSkel* generates sequential and OpenMP code for a CPU, and CUDA code for a GPU; in each single host node, one of the CPU cores is controlling the GPU and the rest of the CPU cores execute the same code that performs the same operation that is being executed on the GPU. In other words, we consider one CPU core (usually the core with rank 0) plus the GPU as one processing element and each other core in the CPU as an independent processing element.

The selection of the GPU card is based on the ID of the OpenMP thread, where the GPU index is derived from the ID to select the GPU. Consequently, all the CUDA kernel calls in our heterogeneous programming model are taking place within the parallel region of the OpenMP code.

5.1.2 *GPU-HWSkel* Characteristics

Besides the characteristics of the base library (*HWSkel*), the *GPU-HWSkel* library provides the following characteristics:

1. **Cost Model:** To ensure a good load balance between the heterogeneous processing elements(GPUs and CPUs) in the system, *GPU-HWSkel* has a new performance cost model (CM2) that take into account the performance capabilities of GPU and CPU for data distribution as discussed in Section 6.2.

2. **Dependencies:** *GPU-HWSkel* uses the CUDA runtime API as a third-party library, therefore all the GPU-HWSkel-based programs that use the *GPU-HWSkel* library need to be compiled by NVCC compiler with the help of a c compiler like gcc.

5.2 Implementing *GPU-HWSkel*

Having presented the implementation principles of the *GPU-HWSkel* library in the previous section, We now discuss the programming model that is used to implement our heterogeneous skeletons in the *GPU-HWSkel* library. Figure 16 shows the underlying target hardware of our approach.

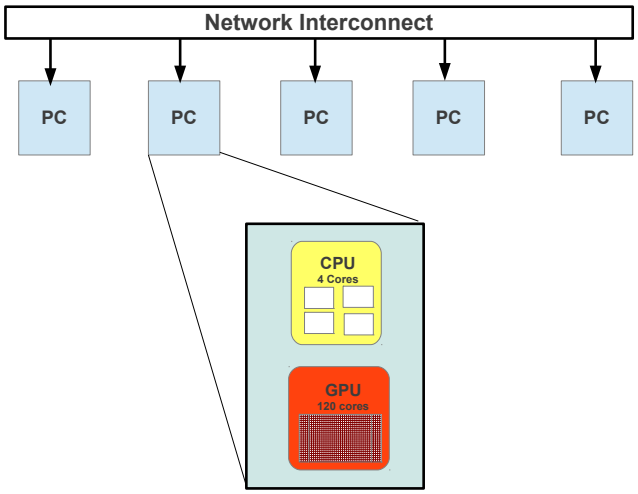


Figure 16: Underlying Hardware of *GPU-HWSkel* Programming Model

Performing heterogeneous computation with our target architecture requires three different types of interaction patterns [160, 161]:

- **Nodes-Interaction** occurs between the nodes within the cluster due to the communication between the processes. This type of interaction can be implemented using any message passing programming model like MPI.

- **Intra-node-Interaction** occurs inside each node among the CPU cores due to threads execution. This interaction requires a shared-memory programming model like OpenMP.

- **CPU-GPU-Interaction** occurs between the CPU and GPU. This interaction requires a library that support all the functions that are needed for transferring data between the CPU and the GPU, and launching the GPU kernels.

As in *HWSkel*, the SPMD programming model that inherited from MPI is used in the top level of our model. In the node-level parallel execution, we use OpenMP for multicore CPU programming and CUDA for GPU programming to provide comprehensive parallel heterogeneous performance comparable with homogeneous multicore performance. Our heterogeneous programming model is illustrated in Figure 17.

The aim of our approach is that the heterogeneous programming model that is used in the *GPU-HWSkel* library will utilise all CPU cores and GPUs in heterogeneous multicore/GPU clusters by using CPU cores and GPU at the same time in each node. The current implementation of our heterogeneous programming model does not allow inter-communications between the GPU and other CPU

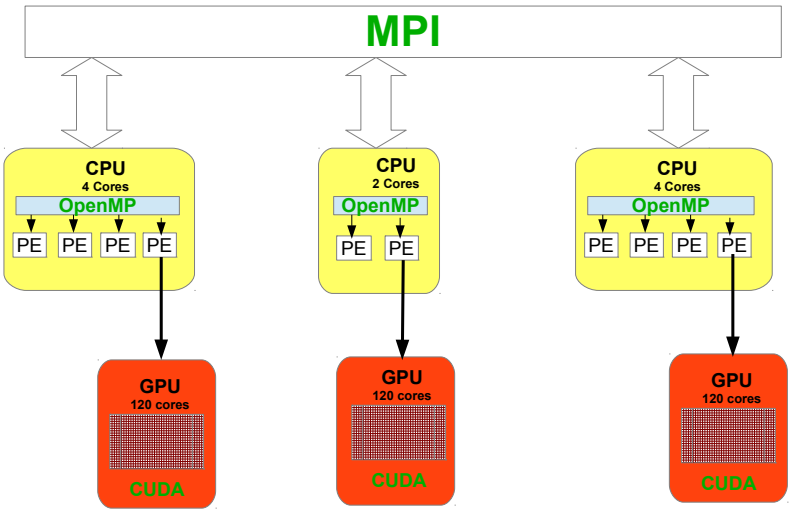


Figure 17: Programming Model of *GPU-HWSkel*

cores inside the host nodes.

5.3 User Functions

The *GPU-HWSkel* library provides programmers with simple user functions that can be passed as arguments to the skeletons. The user functions are implemented using macros to create CUDA kernel code for the GPU as well as C-like functions for CPU execution. For each skeleton in the library we define a corresponding user function. For example, Listing 5.1 displays the macro for the *hMap* user function.

```
1
2 /* MAP user function. */
3
4 MAP_FUNC(square, float, x,
5           return x*x;
6 );
```

Listing 5.1: MAP User Function.

while Listing 5.2 displays the macro expansion.

```
1
2 /*
3  * Macro definition
4  * This macro creates a CUDA kernel & C-like function
5  * for map skeleton.
6  */
7
8 #define MAP_FUNC(funcName, dataType, parameter, funcBody)\
9 __device__ dataType\
10
11 DEV_MAP_FUNC(dataType parameter)\
12 {\
13   funcBody\
14 }\
15 dataType CPU_MAP_FUNC(dataType parameter)\
16 {\
17    funcBody\
18 }
```

Listing 5.2: Macro expansion.

5.4 Skeletons in *GPU-HWSkel*

In the *HWSkel* library we described the skeletons that represent the implementations of MPI and OpenMP. In this section we will discuss the new skeletons versions that represent CUDA kernels for GPU implementation besides C-like function for CPU implementation.

The *GPU-HWSkel* currently provides a set of data-parallel heterogeneous skeletons including `hMap, hMapAll, hReduce, hMapReduce and hMapReduceAll`.

For each skeleton in *GPU-HWSkel*, a general interface is provided to be used

for skeleton invocation, regardless of whether the skeleton is executed on a CPU or GPU. Therefore, during the application development process, the application programmer only needs to specify a well-suited skeleton interface and the appropriate user function for the given problem, and not focus on where the skeleton will be executed. Since the user function is introduced in *GPU-HWSkel* to provide the definition of the function that will be executed by the skeleton, we modified the interface of the previous skeletons by eliminating the parameter that represents the function name.

```
1
2 #include "HwSkel.h"
3
4 REDUCE_FUNC(plus, float,  x, y,
5           return x+y;
6 );
7
8 MAP_FUNC(square, float, x,
9          return x*x;
10 );
11
12 int main (int argc, char **argv)
13 {
14     int len;
15     float *data, *map_result, reduce_result;
16
17     //- Skeleton initialisation
18     InitHWSkel(argc, argv);
19
20     if(StartNode)
21     {
22         sscanf(argv[1],"%d",&len);
23
24         //- allocate memory on host
25         data = (float*)malloc(len*sizeof(float));
26
27         //- fill the array
28         int i;
29         for(i=0;i<len;i++)
30         {
31             data[i] = i+1;
32         }
33      }
34
35      //- call hMap skeleton
36      map_result = hMap(data, len, FLOAT);
37
38      //- call hReduce skeleton
39      reduce_result = hReduce(map_result, len, FLOAT);
40
41      if(StartNode){
```

```
42          printf("C:sum of squar between [ 0 .. %d ] = %.f\n",len,reduce_result);
43        }
44
45        //- skeleton termination
46        TerminationHWSkel();
47
48        return 0;
49  }
```

Listing 5.3: An Example of using hMap and hReduce skeletons with User Functions.

The implementation of our skeletons depends on the available architectures and an automatic-implementation selection plan is used to select the appropriate implementation as we discussed earlier. Thus, if a GPU device is found and a CUDA implementation is needed, the skeleton registers all the property details of the selected device such as the maximum number of threads per block and the maximum number of blocks supported by device, and then sets up the execution configuration and kernel parameters in a transparent way.

Each skeleton has a corresponding user function, Listing 5.3 shows an example of using the map and reduce skeletons along with their corresponding user functions.

5.5 Summary

In this chapter we have presented a skeleton-based programming framework called *GPU-HWSkel* for heterogeneous multicore/GPU architectures including single-core CPU, multicore CPU, GPU, and *integrated* multicore/GPU systems. Our framework is intended to simplify the implementation of parallel applications and

supports execution on heterogeneous multicore/GPU systems (i.e. a cluster of multicore/GPU nodes) by providing high-level heterogeneous parallel skeletons for CPU and GPU programming, which conceal the complexity of parallelisation, in particular, communication between the CPU and GPU.

Chapter 6

A GPU Workload Distribution Cost Model (CM2)

In this chapter we present and discuss an extension CM2 cost model of this approach to account for the GPU as an independent processing element, and to automatically find a good distribution in heterogeneous multicore/GPU systems. Like the CM1 cost model, CM2 is a static cost model, dynamically parametrised to provide performance portability.

Both CPU cores and a GPU device are considered as a single independent processing unit in each host node of the heterogeneous multicore/GPU cluster as we mentioned earlier. Thus we aim to extend our cost model to estimate the relative processing power of a CPU as well as a GPU in each host node to guide the distribution of workload across the cluster nodes, and between the GPU and CPU cores on each host node. Our heterogeneous skeletons will fully automate

the distribution process on a heterogeneous multicore/GPU architecture by using the extended performance cost model, which in turn makes the task of workload distribution much simpler for the skeleton programmer.

This chapter presents our extended CM2 cost model for heterogeneous multicore/GPU systems . We start by reviewing the related performance cost models for distributing workload on heterogeneous architectures that comprise multicore CPU and GPU in Section 6.1. In Section 6.2 we discuss our approach. Then we describe the methodology of building our performance cost model for the distribution of workload across CPU cores and GPU cards in heterogeneous multicore/GPU systems in Section 6.3. Finally we discuss experimental results to evaluate the effectiveness of the CM2 cost model in the performance of the *GPU-HWSkel* heterogeneous skeletons for data parallelism on heterogeneous multicore/GPU systems in Section 6.4.

6.1 Related Work

A number of performance cost models have been developed for heterogeneous parallel systems (See Section 4.1). To the best of our knowledge little research has been done in considering the use of CPU cores and a GPU card simultaneously on heterogeneous multicore/GPU configurations. Here we briefly describe related models that consider using heterogeneous multicore/GPU systems.

- A mathematical performance cost model is introduced in conjunction with

a 2D-FFT library for finding the optimal distribution ratios between CPUs and GPUs in a heterogeneous system [162]. The model is constructed to predict the total execution time of a 2D-FFT of arbitrary data size. Firstly the FFT computation is split into small steps, and then the model predicts the execution time for each execution step using profiling results, and finally the model determines the optimal load distribution ratio as the shortest predicted execution time. Moreover the model attempts to overcome the limitation in the memory sizes of GPUs by iterating GPU library calls.

- An adaptive mapping technique [11] is implemented in the heterogeneous programming system called Qilin (See Section 2.4.2.4) for computation placement on heterogeneous multiprocessors. It is a fully automatic adaptive approach to find the optimal computation mapping to processing elements. Qilin has a capability to use any heterogeneous platform, since it does not require any hardware information for its implementations. This technique uses a execution-time projection stored in a database to determine the execution times of both the CPU and GPU for a given program, problem size and hardware configuration. Further, the determined execution times are used to statically partition the workload among the CPU and GPU. Thus, the first step in the Qilin programming system is to conduct a training run to add data to the database.

- An optimisation framework is introduced in [163] to improve the load balance on heterogeneous multicore/GPU systems. Instead of using static partitioning method, the model applies an adaptive technique that dynamically balances the workload distribution between the CPU cores and the GPU in a single node. At the beginning of the execution process, the model measures the performance of both the CPU and the GPU, and then the measurement is used to guide the workload distribution. In addition to this adaptive technique, the model tries to hide the communication overhead of transferring the data between CPU and GPU by providing software pipelining to overlap data transfers and kernel execution.

6.2 Discussion

Load-balancing at the multi-node heterogeneous multicore/GPU hardware level can either be done dynamically or statically before program execution is started. Static cost models incur less overhead than dynamic models due to their simplicity and lack of runtime overhead. Besides, heterogeneous multicore/GPU systems are highly distributed, and data transfer between host main memory and GPU device memory is costly. So we seek to minimise this before program execution.

Therefore we wish to develop an accurate cost model and prediction mechanism to balance the workload distribution across the cores and the GPU in each

single node as well as between the nodes in a heterogeneous multicore/GPU cluster. The new cost model inherits all the features of the CM1 cost model described in Section 4.3.

However, in contrast to the performance cost models described previously, our new cost model provides the following features.

Heterogeneous-mode. The performance cost models in [162, 11, 163] are based on measuring the performance of both the CPU and the GPU in a heterogeneous system by using profiling results. Our CM2 cost model is based on two different type of performance measurements for the core and GPU. Since the performance of the GPU is changed by changing the data size while the performance of the core can be more stable for different data sizes, we need only measure the performance of the GPU with a training run and the runtime of the program on one reference core of the system, while the other cores performance is calculated using the architectural parameters that were introduced in the CM1 cost model.

Hardware-auto-selection. Since our performance cost model can provide enough information about the CPU and GPU performance capability, therefore, our heterogeneous skeletons can potentially choose to use either the multicore CPU or a GPU card to execute the ongoing program. This feature will be discussed in more detail in the future work in Section 7.3.

6.3 The CM2 Cost Model

We wish to reconstruct our architectural cost model in the *HWSkel* library to distribute the workload on heterogeneous multicore/GPU systems by statically determining the chunk size of data for each processing element either core or GPU in the system.

Therefore, since our underlying target hardware consists of two levels of heterogeneous hardware architectures, the proposed new cost model is viewed as two-phase cost model. The model is divided into two main components: *i)* ***Single-Node Model***, to guide the workload distribution across the CPU cores and the GPU device inside each node in the multicore/GPU system; *ii)* ***Multi-Node Model***, to balance the workload across the nodes in the cluster.

In general, we focus on predicting the runtime of the application code on the GPU device and use the simple CM1 architectural cost model that was used by the *HWSkel* library for measuring the processing power of the multicore CPU. In addition, since the workload is statically distributed across the CPU cores and the GPU and also between the nodes at the beginning of program execution, the model does not allow for any communication between the CPU cores and the GPU or between the nodes in the system other than via the skeleton.

We will now discuss the two phases of our performance cost model in more details.

6.3.1 Single-Node Cost Model

In a heterogeneous multicore/GPU node, the GPU is connected to the multicore CPU via a PCI Express connection. The *Single-Node* cost model is viewed as the method that tunes the distribution of the workload between the CPU cores and the GPU.

In constructing the *Single-Node* cost model, we have assumed that one of the CPU cores (usually the core with rank 0) is dedicated to control and to communicate with the GPU device, and that the rest the of the CPU cores will be executing the same computation. Thus, we assume that the CPU at least has two cores. We also assume that there is no inter-process communication either among the CPU cores or between the GPU and non-dedicated CPU cores.

We base the workload distribution on the performance ratio between the core and GPU in the integrated node. So the cost model aims to predict the execution time of a single core vs. the GPU device for arbitrary data sizes, and calculates the chunk size for a CPU core and the GPU by using this performance ratio.

To facilitate our discussion, let us introduce the following notation:

T_C : Program runtime on a single core.

T_G : Program runtime on the GPU.

P : The relative power of a computational unit, *e.g* a core, GPU.

C : Number of cores in a single node.

D : Data Size.

We start by calculating P, the relative powers of the GPU and a single core:

$$P = T_C/T_G$$

If the GPU is allocated D_{GPU} units of data then the multicore will receive [1]

$$D_{GPU}.P/(C-1)$$

units. As the node comprises a multicore and a single GPU, the total data size is

$$D_{total} = D_{GPU} + D_{GPU}.P/(C-1) \tag{11}$$

Factoring out D_{GPU}, the data allocated to the multicore is

$$D_{multicore} = D_{total}/(1 + P/(C-1)) \tag{12}$$

and the each core is allocated

$$D_{core} = D_{multicore}/(C-1) \tag{13}$$

[1] $C-1$ as one of the cores is dedicated to the GPU.

6.3.2 Multi-Node Cost Model

The *Multi-Node* cost model is based on the *Single-Node* cost model to determine the chunk size for each node in the system. As a heterogeneous cluster might have different kinds of computing nodes, the key idea of the *Multi-Node* cost model is to measure the relative processing power for each node in the cluster. Hence, the total available processing power P for n nodes is given by:

$$P_{total} = \sum_{i=1}^{i=n} P_i$$

So for data size D_{total}, the chunk size for node i is:

$$(P_i/P_{total}).D_{total} \tag{14}$$

Nodes may have different architectures, and hence powers. The relative power of a node i that consists of a GPU and multiple cores is the sum of the relative powers of the cores, P_{core}, and the GPU P_{GPU}:

$$P_i = P_{GPU_i} + (C_i - 1).P_{core_i} \tag{15}$$

if there is only a single core, *i.e.* $C = 1$, it follows directly that

$$P_i = P_{GPU_i} \tag{16}$$

To calculate P_{core} and P_{GPU}, we first measure $T_{C_{base}}$, the runtime of the program on core of the system, and use it as follows:

$$P_{GPU_i} = T_{C_{base}}/T_{G_i} \tag{17}$$

In practice we predict the relative powers on the base core, $P_{C_{base}}$, and on the cores of node i, P_{C_i} using the CM1 cost model, *i.e.* Equation 7 in Section 4.4:

$$P_{C_i} = S_i.L2_i \tag{18}$$

$$P_{C_{base}} = S_{base}.L2_{base} \tag{19}$$

Hence the relative power of a core on node i is:

$$P_{core_i} = P_{C_{base}}/P_{C_i} \tag{20}$$

Substituting equations (17) and (20) in (15) gives the cost equation used in the *GPU-HWSkel* library:

$$P_i = P_{GPU_i} + (C_i - 1).P_{C_i}/P_{C_{base}} \tag{21}$$

The key point is that we need only measure T_{G_i} and $T_{C_{base}}$ to parametrise the model.

6.4 *GPU-HWSkel* Evaluation

Having presented the implementation of the *GPU-HWSkel* framework and the CM2 cost model, we will now discuss and report experimental results to evaluate the achievable performance with our skeletons that exploit CM2. The experiments are conducted to justify and demonstrate the necessity of the CM2 cost model for *GPU-HWSkel*, and also to check the behavioural consistency of the skeletons with CM2 on a number of different parallel architectures that comprise multicore CPU and GPU in each node.

6.4.1 Benchmarks

The experiments are based on running the common Matrix Multiplication parallel kernel, and also a simple Fibonacci program. A brief description of each benchmark application is given in the following sections.

6.4.1.1 Matrix Multiplication

The well-known representative for a wide range of high-performance applications is the problem of multiplying two matrices A with element $a_{m\times n}$ in row m and column n, and B with element $b_{n\times k}$ in row n and column k resulting in the matrix C with element $c_{m\times k}$ in row m and column k. There are a number of different techniques for multiplying matrices. Here we are using matrix multiplication algorithm to evaluate the performance of our heterogeneous Map skeleton.

Sequential Algorithm: In our algorithm the number of multiplications performed is reduced by breaking down the input matrices into several sub-matrices. Thus, to compute the matrix product C = AB,

let:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad and \qquad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where A_{ij} and B_{ij} are sub-matrices of matrix A and matrix B.

Then the matrix C is computed by:

$$C = \begin{bmatrix} A_{11}B_{11} \oplus A_{12}B_{21} & A_{11}B_{12} \oplus A_{12}B_{22} \\ A_{21}B_{11} \oplus A_{22}B_{21} & A_{21}B_{12} \oplus A_{22}B_{22} \end{bmatrix}$$

GPU-HWSkel-based Parallel Algorithm: In the parallel version, the input matrices are broken down into blocks of small matrices where the block size is determined by the programmer, and then the resulting matrix is stored as a array (Map_{array}) of sub-matrices elements to maintain the suitability of using the Map skeleton. Each element in the Map_{array} contains all sub-matrices that are needed to compute one of the sub-matrices in the

resulting matrix as follows:

$$Map_{array} = \boxed{A_{11}B_{11}A_{12}B_{21}} \boxed{A_{11}B_{12}A_{12}B_{22}} \boxed{A_{21}B_{11}A_{22}B_{21}} \boxed{A_{21}B_{12}A_{22}B_{22}}$$

The first step of the algorithm is performed only by the master PE and is not included in the runtime. In the second step the chunk size for each PE is determined using our performance cost model, and the array elements are distributed across the available PEs using the *hMap* skeleton.

6.4.1.2 Fibonacci Program

Fibonacci is a function that computes Fibonacci numbers (See Code fragment 6.1). In our experiment, we use a simple program that calculate the Fibonacci value for an array of integer numbers with fixed constants by replicating the fib function from the original sequential program.

```
1
2  long fib(long n)
3  {
4      long first = 0;
5      long second = 1;
6      long tmp;
7      while (n--){
8          tmp = first + second;
9          first = second;
10         second = tmp;
11     }
12     return first;
13 }
```

Listing 6.1: Code for `Fibonacci` function.

In the parallel version, the array of integers is split into chunks using a split function which employs the cost model for load distribution, and then the fib

function is mapped in parallel across each chunk. Code fragment 6.2 presents the main *Fibonacci* program that uses the *hMapReduce* skeleton.

```
1
2  int main (int argc, char **argv)
3  {
4      int i;
5      int len;
6      long *data;
7      long *map_result;
8
9      //- skeleton initialisation
10     Skeletons_Init(argc, argv);
11
12     if(StartNode)
13     {
14         sscanf(argv[1],"%d",&len);
15
16         //- allocate memory on host
17         data = (long*)malloc(len*sizeof(long));
18
19         //- fill the array
20         for(i=0;i<len;i++)
21           data[i] = 1000000;
22      }
23
24      //- call map skeleton
25      map_result = hMap(data,len,LONG);
26
27      //- skeleton termination
28      Skeletons_Termination();
29      return 0;
30 }
```

Listing 6.2: Main program for *hMap* `Fibonacci`.

6.4.2 Platform

We conduct our experiments on a heterogeneous multicore/GPU cluster of a number of different parallel architectures located at Heriot-Watt University as summarised in Table 8.

- ***lxpara:*** an eight-core Dell PowerEdge 2950 machines constructed from two quad-core Intel Xeon 5410 processors running Linux RedHat 5.5 at 2.33GHz with 6144 KB L2 cache and using 8GB RAM.

- ***lxphd:*** a two-core machines with Intel E8400 processors running RedHat 5.5 at 3.00GHz with 6144 KB L2 cache and using 2GB RAM.

- ***linux_lab:*** a two-core machines consisting of Linux RedHat 4.1.2 workstations with a 2.4GHz Intel processor, using 2GB RAM and 2048KB L2 cache.

- ***brahma:*** a four-core machine running Linux RedHat 4.1.2 at 3.06GHz with 512 KB L2 cache and using 4GB RAM.

Each of the above machines is connected to an NVIDIA GeForce GT 520 GPU device. The device has 1 GB of DRAM and has one multiprocessor (MIMD units) clocked at 810 MHz. The multiprocessor has 48 processor cores (SIMD units) running at twice the clock frequency of the multiprocessor and has 16 KB of shared memory.

Further, CUDA version 4.0 was used for the experiments. The CUDA code was compiled using the NVIDIA CUDA Compiler (NVCC) to generate the device code that is launched from the host CPU.

Machine	**CPU**				**GPU**			
name	**archi**	**Cores**	**MHz**	**L2**	**archi**	**SM**	**Cores**	**MHz**
lxpara	Xeon 5410	8	1998	6144KB	GT 520	1	48	1620
lxphd	Intel E8400	2	1998	6144KB	GT 520	1	48	1620
linux_lab	2 Duo CPU	2	1200	2048KB	GT 520	1	48	1620
brahma	Xeon(TM)	4	3065	512KB	GT 520	1	48	1620

Table 8: Experimental Architectures.

6.4.3 Performance Evaluation

In our experiments, we investigate the performance impact of the heterogeneous *hMap* skeleton on different heterogeneous parallel architectures (outlined in Section 6.4.2) including multicore CPU and GPU systems. To ensure the effectiveness and the portability of our performance cost model, we have carried out our experiments on two modes of parallel architecture: *i)* ***Single-Node***, where the hardware comprises a single multicore CPU connected to a single GPU device; *ii)* ***Multi-Node*** or cluster, where the cluster consists of a number of loosely-coupled nodes with a multicore processor and a single GPU device.

For all the measurements that are performed on the two-core processors (such as *linux* and *lxphd*) , we follow the common practice of increasing the input-data size to evaluate the behaviour consistency of the *hMap* skeleton with the CM2 cost model. While in the case of using machines with an eight-core processor (such as *lxpara*), all programs are measured with a fixed data size on 1,2,3,4,5,6, and 7 cores together with a single GPU device. We measure the runtimes on *lxpara* for the *hMap* skeleton implementation, with a fixed data size of 1500 x 1500 for the input matrices, and 80,000 elements of Fibonacci (1,000,000).

6.4.3.1 Single Multicore/GPU Node Results

The single-node experiments have been carried out our on *linux_lab*, *lxphd*, and *lxpara* as single nodes.

Table 9 and 10 show the *hMap* runtime for Matrix Multiplication and Fibonacci on *linux_lab* and *lxphd* respectively. The measurements report the runtime on 1 core, GPU, GPU plus 1 core, and show the percentage improvement of *hMap* using the CM2 cost model. The *hMap* Fibonacci has an improvement of 95% over the sequential time and improvement of 4% over the GPU time on *linux_lab* and *lxphd* using CM2 , while the *hMap* Matrix Multiplication has an improvement of 68% over the sequential time on both *linux_lab* and *lxphd*, and improvement of 32% on *linux_lab* and 20% over the GPU time on *lxphd*.

Data Size	Run-Time (s)			1 Core+GPU Improvement %	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
800x800	2.31	1.40	1.32	42%	5%
900x900	3.30	1.77	1.54	53%	12%
1000x1000	4.52	2.09	1.80	60%	13%
1100x1100	6.02	2.73	2.12	64%	22%
1200x1200	7.82	3.26	2.54	68%	22%
1300x1300	9.94	4.29	3.19	67%	25%
1400x1400	12.41	5.37	4.00	67%	25%
1500x1500	15.26	7.23	4.91	67%	32%

(a) Matrix Multiplication

Data Size	Run-Time (s)			1 Core+GPU Improvement %	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
1000	3.36	0.19	0.17	94%	10%
2000	6.77	0.34	0.32	95%	5%
5000	17.02	0.79	0.75	95%	5%
10000	34.17	1.53	1.47	95%	3%
20000	67.93	3.06	2.91	95%	4%
30000	103.30	4.55	4.39	95%	3%
40000	137.08	6.071	5.79	95%	4%
50000	170.80	7.55	7.24	95%	4%
60000	207.33	9.05	8.69	95%	4%
70000	243.79	10.51	10.12	95%	3%
80000	278.04	12.05	11.52	95%	4%

(b) Fibonacci

Table 9: 1 Core *hMap* Runtimes (*linux_lab*).

Data Size	Run-Time (s)			1 Core+GPU Improvement %	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
800x800	4.28	1.47	1.41	67%	4%
900x900	6.09	1.84	1.66	72%	9%
1000x1000	8.37	2.25	1.98	76%	12%
1100x1100	11.12	2.88	2.36	78%	18%
1200x1200	14.43	3.49	3.07	78%	12%
1300x1300	18.34	4.46	3.90	78%	12%
1400x1400	22.91	5.79	4.88	78%	12%
1500x1500	28.25	7.61	6.02	78%	20%

(a) Matrix Multiplication

Data Size	Run-Time (s)			1 Core+GPU Improvement %	
	1 Core	GPU	1 Core+GPU	1 Core	GPU
1000	3.27	0.20	0.19	94%	5%
2000	6.53	0.36	0.34	94%	5%
5000	16.36	0.79	0.77	95%	2%
10000	32.75	1.55	1.48	95%	4%
20000	65.47	3.07	2.93	95%	4%
30000	98.13	4.55	4.39	95%	3%
40000	130.97	6.07	5.77	95%	4%
50000	163.57	7.53	7.22	95%	4%
60000	196.44	9.06	8.67	95%	4%
70000	229.18	10.55	10.06	95%	4%
80000	261.77	12.00	11.52	95%	4%

(b) Fibonacci

Table 10: 1 Core *hMap* Runtimes (*lxphd*).

Table 11 shows the runtime of Matrix Multiplication with data size of 1500 x 1500 and Fibonacci with data size 80,000 elements with a value of 1,000,000 using *hMap* on *lxpara*. The measurements show that the *hMap* Fibonacci has improvement of 77% over 8 cores, while the *hMap* Matrix Multiplication shows that there is no improvement after 6 cores.

Cores	**Run-Time (s)**			**(Core-1)+GPU Improvement %**	
	Cores	**GPU**	**(Cores-1)+GPU**	**Cores**	**GPU**
1	19.60	7.26	7.26	62%	0%
2	9.82	7.26	5.31	45%	26%
3	6.55	7.26	4.20	35%	42%
4	4.93	7.26	3.48	29%	52%
5	3.94	7.26	3.09	21%	57%
6	3.29	7.26	2.92	11%	59%
7	2.89	7.26	2.84	1%	60%
8	2.54	7.26	2.78	-9%	61%

(a) Matrix Multiplication

Cores	**Run-Time (s)**			**(Core-1)+GPU Improvement %**	
	Cores	**GPU**	**(Cores-1+GPU)**	**Cores**	**GPU**
1	344.37	12.03	12.03	96%	0%
2	172.01	12.03	11.60	93%	3%
3	114.85	12.03	11.28	90%	6%
4	86.06	12.03	10.90	87%	9%
5	68.99	12.03	10.59	84%	11%
6	57.43	12.03	10.27	82%	14%
7	49.26	12.03	9.96	79%	17%
8	43.09	12.03	9.69	77%	19%

(b) Fibonacci

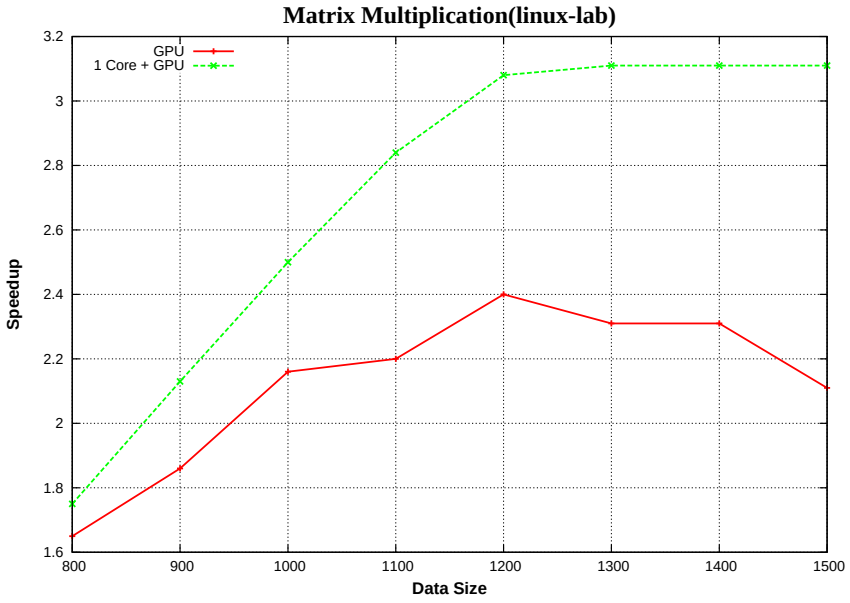
Table 11: Multiple Core *hMap* Runtimes (*lxpara*).

The parallel performance is measured as the absolute speedup of using both the GPU and the cores within a single machine. Figures 18 and 19 show the absolute speedup achieved for the Fibonacci and Matrix Multiplication programs with different input-data sizes on the two-core *linux* and *lxphd* machines respectively. As anticipated, our results show that using CPU cores together with a GPU to perform the same computing operations concurrently can improve the performance of our skeletons.

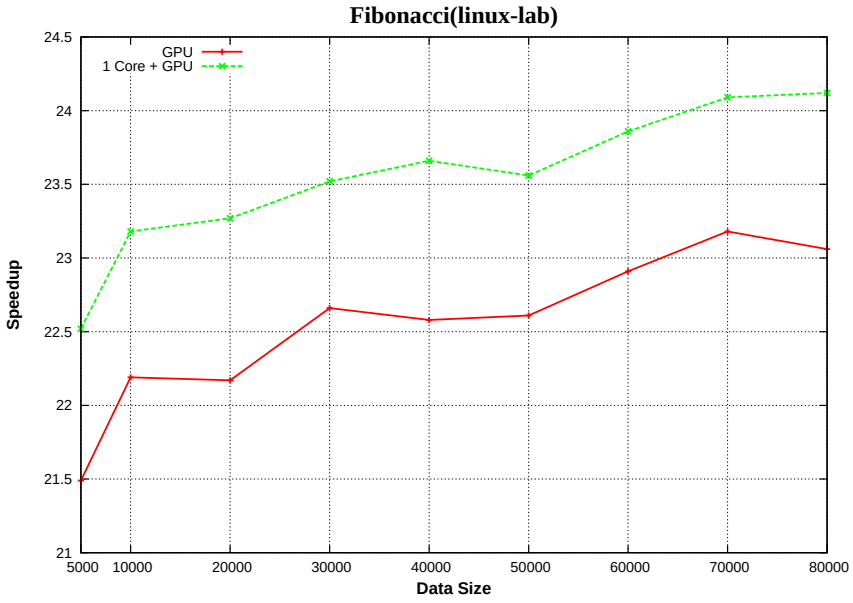
The graphs in Figures 18 and 19 compare the absolute speedup curve for (one CPU-core plus single GPU) implementation with the curve for GPU implementation. Although the computing capability of the GPU is relatively large compared with the computational power of a single CPU-core, the results show that using cores together with a GPU can deliver expected and acceptable speedups on both machines.

Our results also suggest that using the performance cost model for determining granularity and data placement on different heterogeneous architectures can provide a good load balance for data distribution between cores and a GPU, for data parallel programs with limited irregularity. This is reflected in the speedup graphs where the curves are broadly similar for both programs (*Fibonacci and Matrix Multiplication*) with different input data size on different parallel heterogeneous architectures.

Next, to investigate the impact of the data distribution strategy that is used in the CM2 cost model on the parallel performance of a varying number of cores

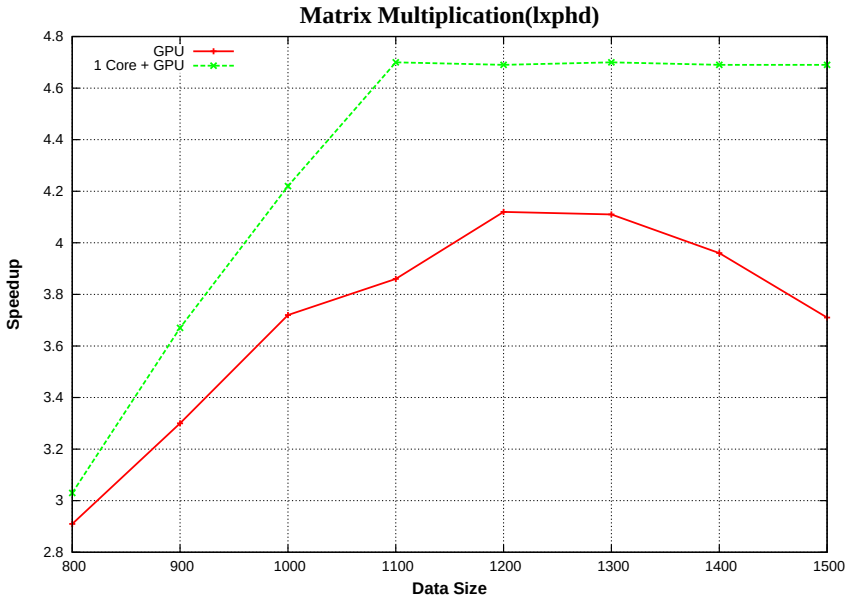


(a)

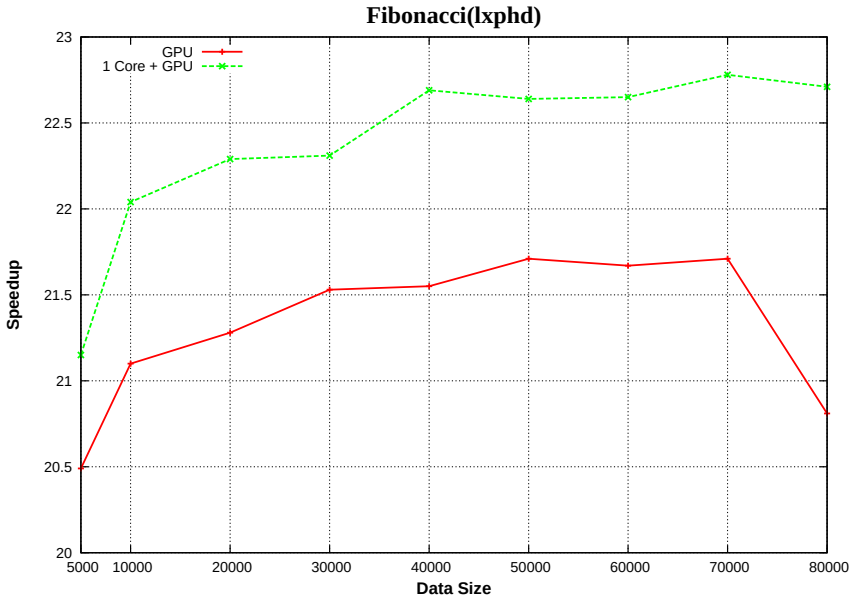


(b)

Figure 18: *hMap* Fibonacci & *hMap* Matrix Multiplication Absolute Speedup on (*linux_lab*)

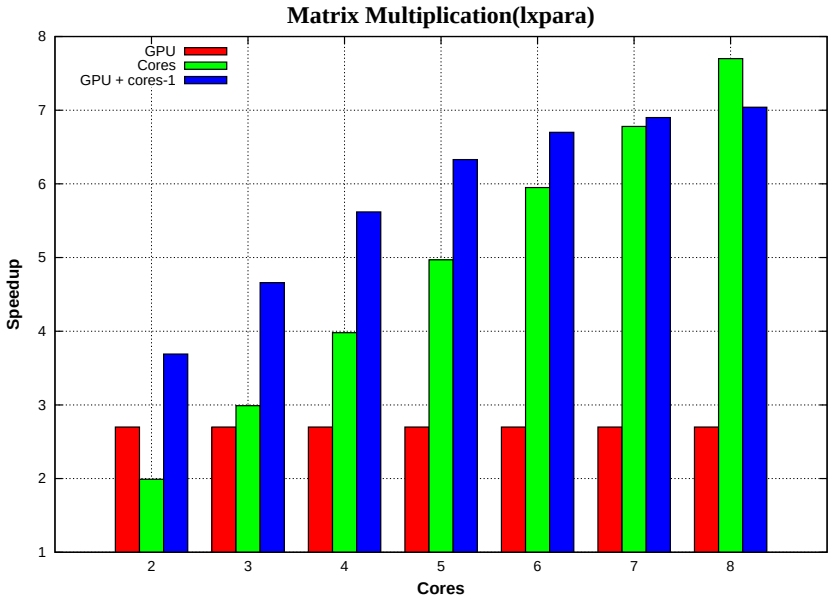


(a)

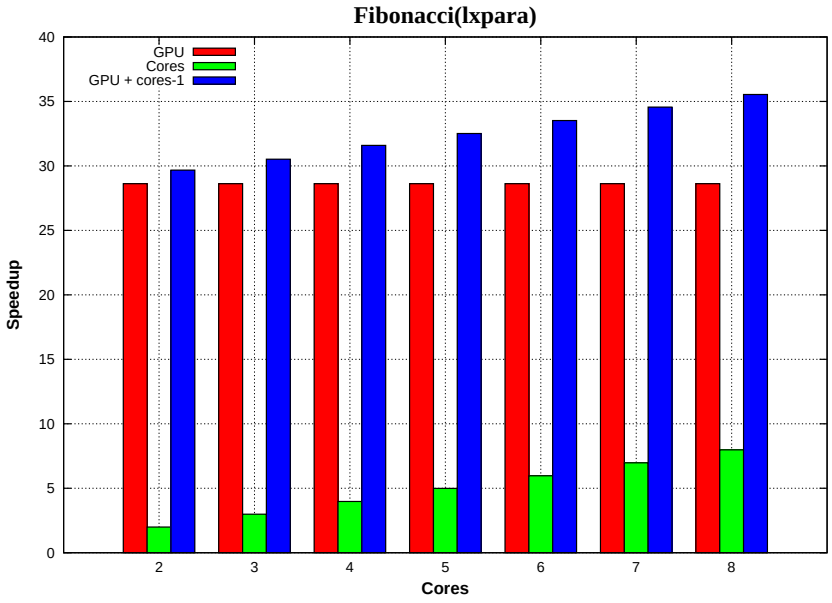


(b)

Figure 19: *hMap* Fibonacci & *hMap* Matrix Multiplication Absolute Speedup on (*lxphd*)



(a)



(b)

Figure 20: *hMap* Fibonacci & *hMap* Matrix Multiplication Absolute Speedup on (lxpara)

with a single GPU-device, we run experiments with the Fibonacci and Matrix Multiplication programs on a machine with eight cores (i.e. *lxpara*).

Figure 20 compares the absolute speedups of both Fibonacci and Matrix Multiplication programs on only *cores* and *GPU*, and *CPU-core+GPU* of *lxpara*. This shows that in both programs we obtain good performance as anticipated. Firstly the results presented in Figure 20 are consistent with others that we obtained for both programs on the *linux* and *lxphd* machines, where the speedup is increased by using one CPU-core plus the GPU. Secondly we have obtained almost linear speedup with parallel efficiency of about 99% in both programs on cores. However, in the Matrix Multiplication program we can see that the speedup has a slight degradation to 95% parallel efficiency after six cores due to decreasing the chunk size to the point that other overheads become more significant. The results show that our skeleton delivers 28x from the GPU compared to a single CPU-core in the Fibonacci program, while we report nearly 2.8x speedup over a CPU-core by using a GPU in the Matrix Multiplication application. The variation in speedup between both programs is due to the GPU-HWSkel-based parallel algorithm used for each program. Since the major problem with GPU implementations which affects the performance efficiency is the size of data being transferred between the CPU and GPU, Strassen's algorithm requires too much data communication between the CPU and GPU, which increases the CPU-GPU communication overhead. Therefore, we find that Strassen's algorithm is more suitable for multicore processors than for a GPU implementation, while the Fibonacci program makes

a good GPU program.

Finally, the results show that the speedup in both programs can be significantly increased by increasing the number of cores along with the help of the GPU. Consequently, using cores together with a GPU can provide an effective way and a sensible environment to execute all programs that are suitable for GPU devices or a multicore processor.

6.4.3.2 Clusters of Multicore/GPU Nodes Results

We now discuss the results of using our performance cost model for data placement on a heterogeneous cluster where each node has a single GPU card connected to a processor with different numbers of CPU cores.

We evaluate the performance of our cost model and its effect on our hMap heterogeneous skeletons by running the skeletons on different combinations of the architectures that are described in Section 6.4.2. Figure 21 plots the speedups for different configurations with different processing elements calculating Fibonacci(1000000) 1500,000 times. The graph compares the speedups of three different kinds of computing units (i.e. cores, GPU, and GPU plus cores) on different numbers of given machines.

Figure 21 shows that the results are consistent with those that were presented in Section 6.4.3.1 where we obtain noticeable speedup by using only the GPU in each host node over using only the cores in the same nodes with different numbers of heterogeneous machines. However, we have improved the performance of our

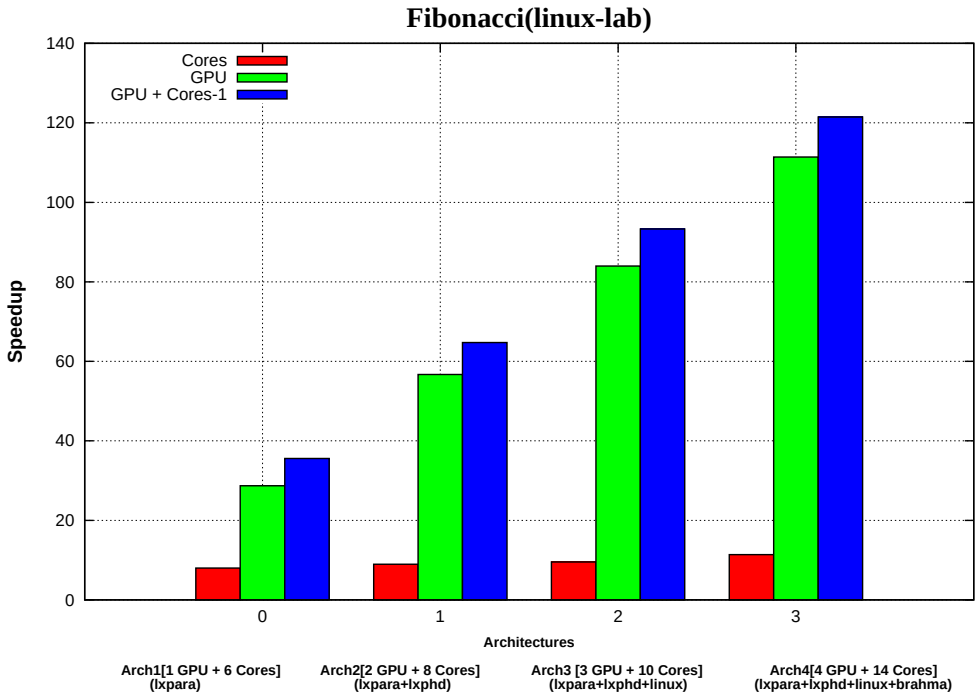


Figure 21: Speedups for the hMap Skeleton on a Heterogeneous Cluster

hMap skeleton by exploiting the cores along with the GPU in each host node.

We suggest once again, that our performance cost model has provided a good strategy of data placement for heterogeneous architectures. The graph shows that the implementation of our hMap skeleton can deliver good scalability, where the upper speedup curve shows improved performance results for using our cost model for data placement between the heterogeneous nodes as will as within each node between the cores and the GPU.

In summary, our experimental results show that using cores together with

a GPU in the same host with our skeleton and cost model can deliver good performance either on a single node architecture or on a multiple nodes (cluster) architecture.

6.5 Summary

In this chapter we present and discuss the CM2 cost model for heterogeneous multicore/GPU systems. The purpose of the new cost model is to balance the workload distribution between the nodes on heterogeneous multicore/GPU cluster as well as between CPU cores (i.e. Equations 14 and 21) and GPU inside each node in cluster (i.e. Equation 12). Our cost model is viewed as two-phase, the *Single-Node* phase to guides workload distribution across CPU core and GPU device using the performance ratio between the CPU and GPU in the multicore/GPU computing node, the *Multi-Node* phase balances the distribution of workload among the nodes on a heterogeneous multicore/GPU cluster. In general, we focus on predicting the runtime of the application code on GPU and use an architectural performance cost model for measuring the processing power of the CPU to calculate the performance ratio.

We have demonstrated how the proposed performance cost model can be integrated in our skeletons to provide an effective static load-balancing strategy with dynamic parametrisation on a heterogeneous multicore/GPU platform, and make the task of workload distribution much simpler for the skeleton programmer.

Finally, we have investigated the feasibility and the necessity of the CM2 cost model to improve the performance of our heterogeneous skeleton using two data parallel benchmarks (See Section 6.4.1) on three single heterogeneous multicore/GPU node architectures and four clusters of heterogeneous multicore/GPU nodes (See Table 8). We show that exploiting both multicore and GPU components of the architecture improve performances in all cases.

Chapter 7

Conclusion

7.1 Introduction

In this thesis, we have addressed the problem of designing an efficient high-level parallel programming model to assure performance portability on heterogeneous parallel systems.

The thesis presented based-skeleton C libraries to simplify parallel programming on heterogeneous parallel architectures including CPUs and GPUs. These libraries provide application developers with heterogeneous parallel skeletons for data parallel computations. In order to achieve an optimal performance on a heterogeneous parallel architecture, performance cost models were provided for the skeletons to explicitly and statically guide the workload distribution on heterogeneous systems.

Chapter 1 discussed the challenges of designing a high-level parallel programming model to abstract all the parallel activities involved in developing parallel applications on heterogeneous systems.

Chapter 2 gave a survey of parallel computing in general and skeleton-based programming in particular, describing existing skeleton frameworks.

Chapter 3 presented a C based skeleton library called *HWSkel* for parallel programming on heterogeneous parallel architectures. It also described and discussed the parallel implementations of the heterogeneous skeletons provided by the library.

Chapter 4 introduced an architecture-aware performance cost model to be used by *HWSkel* skeletons to guide data distribution over a heterogeneous multicore cluster.

Chapter 5 presented the *GPU-HWSkel* library that provides support for parallel programming on either CPU cores or a GPU in heterogeneous multicore/GPU systems.

Chapter 6 presented and evaluated an extension cost model (CM2) of the CM1 cost model.

7.2 Contributions of Thesis

This thesis makes the following contributions in the area of high-level parallel programming models in general and skeleton-based parallel programming and

performance cost models in particular:

- A skeleton-based parallel programming framework named *HWSkel* has been presented as a base library to provide support for parallel programming on heterogeneous multicore cluster architectures. This library is implemented in C on top of MPI as a distributed-memory programming model and OpenMP for shared-memory parallelism. This means that the heterogeneous skeletons can take advantage straightforwardly of its underlying hybrid programming model to be executed either on distributed-memory systems, shared-memory systems or distributed-shared memory architectures. In particular, the *HWSkel* framework provides a set of heterogeneous skeletons for data parallel computations such as *hMap, hReduce, hMapReduce, and hMapReduceAll*. The *HWSkel* framework also provides wrapper functions for MPI routines to keep the user away from using a new programming language within the skeletal programs.

- The second contribution of this thesis is a performance cost model (CM1) to improve the performance of *HWSkel* skeletons on systems composed of heterogeneous multicore nodes. The model was integrated into *HWSkel* to provide a load-balancing strategy in a transparent way. The cost model is architecture-aware and is used to statically determine the data-chunk size according to the number of cores, clock speed and crucially the cache size for each node across a heterogeneous multicore cluster. The cost model

supports performance portability by providing cost estimations on a broad range of heterogeneous platforms.

- The third contribution is designing an extended library named *GPU-HWSkel*. The library provides a high-level parallel programming environment through skeletons to program systems including (single- and multicore) CPU, and GPU architectures. As in the *HWSkel* base library, at the top level, MPI is used to communicate between the nodes, OpenMP is employed in the second level for multicore programming, and the last level is implementing GPU programming by using CUDA. The *GPU-HWSkel* framework also provides the programmers with an simple user functions, using macros to create CUDA kernels code for GPUs as well as C-like functions.

- The fourth contribution is an extension (CM2) to the CM1 cost model to balance the workload distribution between the nodes in multicore/GPU systems as well as between the CPU cores and the GPU inside each node. The model is viewed as two-phase since the underlying target hardware consists of two level of heterogeneous hardware architectures: *i) Single-Node Phase*, to guide the workloads distribution across the CPU cores and the GPU inside each node in the multicore/GPU system; *ii) Multi-Node Phase*, to balance the workload across the nodes in the cluster.

7.3 Limitations and Future Work

The work in this thesis has a number of limitations. This section discusses the main limitations and the possible solutions.

7.3.1 Distributed Data Structures

As discussed in Section 3.2.1 the data structures need to be packed into an *ArrayList* before the distribution process and then unpacked on each processor. Currently, the packing and unpacking are done explicitly by the user. However, both the packing and unpacking procedures can be done automatically by introducing new constructs, where the user can describe the data structures in abstract way like user defined MPI data type.

7.3.2 Exploring Multiple GPUs

With the growth in heterogeneity, where a multicore is coupled with multi GPUs, skeletons are needed to handle all the challenges that introduced by such architecture. Moreover the current implementation of *GPU-HWSkel* does not provide support to multi-GPU systems.

Nevertheless *GPU-HWSkel* can potentially exploit as many GPUs as are available in the system. As we described in Section 5.2, each GPU will be controlled by one of the CPU cores, so, the number of GPUs that can be used in the system is limited to the number of cores in the system. Figure 22 shows how multi-GPU

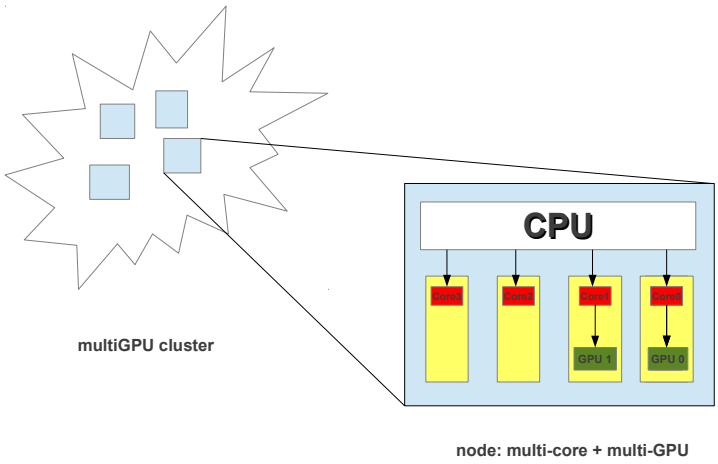


Figure 22: Multi-GPU Support in *GPU-HWSkel*

computing can be managed in each node within a heterogeneous multicore/multi-GPU clusters. Thus, to make multi-GPU implementations possible, each CPU core is connected to one device in the host node. Each CPU core ID is used to set the GPU index.

7.3.3 New Skeletons

Currently, *HWSkel* and *GPU-HWSkel* frameworks provide heterogeneous skeletons that only support mostly regular data-parallel computations on heterogeneous architectures. An important future work would be the extension of our framework by adding new heterogeneous skeletons for task-parallel computations as well as more skeletons for data-parallel computations, and this also could allow for task- and data-parallel skeleton integration.

7.3.4 Automatic Configuration

The key point of designing our framework is to provide applications developers with heterogeneous skeletons that are smart enough to expose the target architecture to invoke the appropriate programming model. Besides, we show in our experiments that some applications may be able to obtain good performance on particular architectures. An interesting topic of future work would be to develop the current heterogeneous skeletons to be more intelligent to expose the target and choose the appropriate hardware and programming for a given problem.

7.3.5 New Model Parameters

We demonstrated that it is possible to obtain good performance using the CM2 cost model for workload distribution with our skeletons on a heterogeneous platform. However, including further architectural parameters (*e.g.* network communication cost) in our cost model has the potential for more accurate estimation and performance improvement on highly heterogeneous environments.

7.3.6 New Platforms

Current implementation of the *GPU-HWSkel* framework are based on the CUDA standard as the backend for GPU programming which limits the implementation of our skeletons to NVIDIA GPU architectures. This should be extended

to cover a broad range of GPU architectures to increase the degree of implementation portability. For example, *GPU-HWSkel* could take advantage of using the OpenCL standard to make other GPU devices accessible through skeleton programming.

Appendix A

The *HWSkel* Library

This appendix presents the complete code for the *HWSkel* Library.

A.1 *hMap Skeleton*

This section presents the *hMap* function discussed in Section 3.2.3.

A.1.1 *hMap*

```
1
2 #include "HwSkel.h"
3
4 //- hMap Skeleton
5
6 void* hMap(void* dataList,int size,enum DataType dType,void* mapFunc)
7 {
8     int i;              // index
9     int* listOfChunks; // list of chunks
10    void *sublist;
11    void *map_result;
12    void **skel_result;
13
14    //- MPI_Datatype Conversion
15    ty = dataTypeConversion(dType);
16
17    if (StartNode)
18    {
```

```
19          //- Master node
20          printf(" hMap Skeleton \n");
21          int dest;
22          int offset;
23
24          skel_result = (void**) malloc(size*sizeof(void*));
25
26          //- call CM1 cost model
27          listOfChunks = CM1_CostModel(currentCluster,size);
28
29          //- sending the chunks size
30          MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
31
32          //- get master's specifications
33          ModelParas *masterInfo = ArrayList_GetItem(currentCluster,id);
34
35          //- master split and send data to workers
36          for(dest=1;dest<np;dest++){
37              sublist = splitCostModel(dataList,dest,listOfChunks);
38              MPI_Send(sublist, listOfChunks[dest], ty, dest, 0, MPI_COMM_WORLD);
39          }
40
41          //- master get own data
42          sublist = splitCostModel(dataList,id,listOfChunks);
43
44          //- master perform its task
45          if(masterInfo->numOfCores > 1){
46              //- call multi-core hMap
47              map_result = MultiCorehMap(sublist,listOfChunks[id],mapFunc);
48              memcpy( skel_result,map_result,listOfChunks[id]*sizeof(void*));
49              offset = listOfChunks[id];
50          }
51          else{
52              //- call single-core hMap
53              map_result = SingleCorehMap(sublist,listOfChunks[id],mapFunc);
54              memcpy( skel_result,map_result,listOfChunks[id]*sizeof(void*));
55              offset = listOfChunks[id];
56          }
57
58          //-  recieve data from the workers
59          for(dest=1;dest<np;dest++){
60              MPI_Recv(map_result, listOfChunks[dest], ty, dest, 0, MPI_COMM_WORLD,
                    &status);
61              memcpy( skel_result+offset,map_result,listOfChunks[dest]*sizeof(void*)
                    );
62              offset+=listOfChunks[dest];
63          }
64          return skel_result;
65      }
66      else //---- Workers -----//
67      {
68          ModelParas *workerInfo = GetProcInfo();
69
70          //- list of chunks
71          listOfChunks = (int*)malloc(np*sizeof(int));
72          MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
73
74          sublist = (void**) malloc(listOfChunks[id]*sizeof(void*));
75          MPI_Recv(sublist, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD, &status);
76
77          if(workerInfo->numOfCores > 1){
78              //- call multi-core hMap
79              map_result = MultiCorehMap(sublist, listOfChunks[id], mapFunc);
80              //-worker send result
81              MPI_Send(map_result, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD);
```

```
82          }
83          else{
84              //- call single-core hMap
85              map_result = SingleCorehMap(sublist,listOfChunks[id],mapFunc);
86              //- worker send result
87              MPI_Send(map_result, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD);
88          }
89      }
90  }
```

Listing A.1: *hMap* Skeleton Code.

A.1.2 *hMap* Single-Core

```
1
2  //- hMap SingleCore
3
4  void* SingleCorehMap(void* dataList,int size, void* funcName)
5  {
6    void *result;
7    //- casting the function pointer
8    void_fp = (void_pFun)funcName;
9    result = (void*)(*void_fp)(dataList,size);
10   return result;
11 }
```

Listing A.2: *hMap SingleCore* Skeleton Code.

A.1.3 *hMap* Multi-Core

```
1
2  //-  hMap MultiCore
3
4  void* MultiCorehMap(void* dataList,int size,void* mapFunc)
5  {
6      void *sublist;
7      void *map_result;
8      void **results;
9      int offset;
10     int myChunkSize;
11     int tid;
12
13     //- get number of threads
14     int threads = omp_get_max_threads();
15
16     //- calculate the chunk size
17     int chunkSize = size/threads;
18
19     //- remainder
20     int remainder = size%threads;
21     results = (void**)malloc(size*sizeof(void*));
22
```

```
23      //-parallel region
24    #pragma omp parallel private(tid,sublist,map_result,myChunkSize,offset) shared
          (threads,dataList,results,chunkSize,remainder)
25    {
26      //- get thread Id
27      tid = omp_get_thread_num();
28
29      if(tid == 0){
30        myChunkSize = chunkSize+remainder;
31        //- partitioning
32        sublist = splitEqual(dataList, myChunkSize, remainder, tid);
33        map_result = SingleCore(sublist, myChunkSize, mapFunc);
34        offset = 0;
35      }
36      else{
37        myChunkSize = chunkSize;
38        //- partitioning
39        sublist = splitEqual(dataList, chunkSize, remainder, tid);
40        map_result = SingleCore(sublist, chunkSize,mapFunc);
41        offset = tid*myChunkSize+remainder;
42      }
43      #pragma omp critical(update_Results)
44      {
45        memcpy(results+offset,map_result,myChunkSize*sizeof(void*));
46      }
47    }
48
49    return results;
50
51 }
```

Listing A.3: *hMapMultiCore* Skeleton Code.

A.2 *hMapAll* Skeleton

This section presents the *hMapAll* function discussed in Section 3.2.4.

A.2.1 *hMapAll*

```
1
2 #include "HwSkel.h"
3
4 //- hMapAll skeleton
5
6 void* hMapAll(void* dataList,int size,enum DataType dType,void* mapFunc)
7 {
8     void *sublist;
9     void *map_result;
10    void **skel_result;
11
12    //- MPI_Datatype Conversion
13    ty = dataTypeConversion(dType);
```

```
14
15      if(StartNode)
16      {
17          printf("  hMapAll Skeleton \n");
18          int dest;
19          int offset;
20
21          //- calculate the chunks size using the CM1 cost model
22          listOfChunks= CM1_CostModel(currentCluster,size);
23
24          //- sending the chunks size
25          MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
26
27           //- list size
28           MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
29
30          //-get master's specifications
31          ModelParas* masterInfo = ArrayList_GetItem(currentCluster,id);
32
33          skel_result = (void**) malloc(size*sizeof(void*));
34
35          //-master distribute the data
36          MPI_Bcast(dataList, size, ty, 0, MPI_COMM_WORLD);
37
38          //-master get own data
39          sublist = splitCostModel(dataList,id,listOfChunks);
40
41          //- master perform its task
42          if(masterInfo->numOfCores > 1){
43               //- call multi-core hMapAll
44               map_result = MultiCorehMapAll(dataList,size,sublist,listOfChunks[id
                     ],mapFunc);
45               memcpy( skel_result,map_result,listOfChunks[id]*sizeof(void*));
46               offset = listOfChunks[id];
47          }
48          else{
49               //- call single-core hMapAll
50               map_result = SingleCorehMapAll(dataList,size,sublist,listOfChunks[
                     id],mapFunc);
51               memcpy( skel_result,map_result,listOfChunks[id]*sizeof(void*));
52               offset = listOfChunks[id];
53          }
54
55          //-master recieve data from the workers
56          for(dest=1;dest<np;dest++){
57              MPI_Recv(map_result, listOfChunks[dest], ty, dest, 0, MPI_COMM_WORLD
                     , &status);
58              memcpy( skel_result+offset,map_result,listOfChunks[dest]*sizeof(void
                     *));
59              offset+=listOfChunks[dest];
60          }
61          return skel_result;
62      }
63      else
64      {
65         //- Workers
66         ModelParas* workerInfo = GetProcInfo();
67
68         //- list of chunks
69         listOfChunks = (int*)malloc(np*sizeof(int));
70
71         //- receive list of chunks
72         MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
73
74         //- receive data size
```

```
75          MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
76          MPI_Bcast(dataList, size, ty, 0, MPI_COMM_WORLD);
77
78          //- split data
79          sublist = splitCostModel(dataList,id,listOfChunks);
80
81          if(workerInfo->numOfCores > 1){
82               //- call multi-core hMapAll
83               map_result = MultiCorehMapAll(dataList,size,sublist,listOfChunks[id
                      ],mapFunc);
84               MPI_Send(map_result,listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD);
85          }
86          else{
87               //- call single-core hMapAll
88               map_result = SingleCorehMapAll(dataList,size,sublist,listOfChunks[id
                      ],mapFunc);
89               MPI_Send(map_result,listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD);
90          }
91     }
92 }
```

Listing A.4: *hMap* Skeleton Code.

A.2.2 *hMap* Single-Core

```
1
2 //- SingleCoreAll
3
4 void* SingleCorehMapAll(void* list,int size, void* sublist, int subsize, void*
      funcName)
5 {
6   void *result;
7   //-casting the function pointer
8   void_fp = (void_pFun)funcName;
9   result=(void*)(*void_fp)(list,size,sublist,subsize);
10  return result;
11 }
```

Listing A.5: *hMapAll SingleCore* Skeleton Code.

A.2.3 *hMap* Multi-Core

```
1
2 //- hMapAll MultiCore
3
4 void* MultiCorehMapAll(void* dataList,int size,void* subDataList,int subsize,
      void* mapFunc)
5 {
6     void *sublist;
7     void *map_result;
8     void **results;
9     int offset;
10    int myChunkSize;
```

```
11      int tid;
12      int threads = omp_get_max_threads();
13      int chunkSize = subsize/threads;
14      int remainder = subsize%threads;
15      results = (void**)malloc(subsize*sizeof(void*));
16      //-parallel region
17      #pragma omp parallel private(tid,sublist,map_result,myChunkSize,offset)
            shared(threads,dataList,subDataList,results,chunkSize,remainder)
18      {
19          tid = omp_get_thread_num();
20          if(tid == 0){
21              myChunkSize = chunkSize+remainder;
22              sublist = splitEqual(subDataList, myChunkSize, remainder, tid);//-
                    partitioning
23              map_result = SingleCoreAll(dataList, size, sublist, myChunkSize,
                    mapFunc);
24              offset = 0;
25          }
26          else{
27               myChunkSize = chunkSize;
28               sublist = splitEqual(subDataList, chunkSize, remainder, tid);//-
                     partitioning
29               map_result = SingleCoreAll(dataList, size, sublist, chunkSize,
                     mapFunc);
30               offset = tid*myChunkSize+remainder;
31          }
32          #pragma omp critical(update_Results)
33          {
34              memcpy(results+offset,map_result,myChunkSize*sizeof(void*));
35          }
36      }
37      return results;
38  }
```

Listing A.6: *hMapAll MultiCore* Skeleton Code.

A.3 *hReduce* Skeleton

This section presents the *hReduce* function discussed in Section 3.2.5.

A.3.1 *hReduce*

```
1
2  #include "HwSkel.h"
3
4  //- hReduce Skeleton
5
6  void* hReduce(void* dataList,int size,enum DataType dType, void* reduceFunc)
7  {
8      int i;              // index
9      int* listOfChunks; // list of chunks
10     void *sublist;
11       void *reduce_result;
12       void **skel_result;
```

```
13
14      //- MPI_Datatype Conversion
15      ty = dataTypeConversion(dType);
16
17      if (StartNode)
18      {
19      printf("  Reduce Skeleton \n");
20        int dest;
21          void **results;
22          results = (void**) malloc(np*sizeof(void*));
23
24          //-calculate the chunks size using cost model
25          listOfChunks = HybridCostModel(currentCluster,size);
26
27          MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPLCOMM_WORLD);
28
29          //-get master's specifications
30          ModelParas *masterInfo = ArrayList_GetItem(currentCluster,id);
31
32         //-master split and send data to workers
33       for(dest=1;dest<np;dest++){
34              sublist = splitCostModel(dataList,dest,listOfChunks);// partitioning
35              MPI_Send(sublist, listOfChunks[dest], ty, dest, 0, MPLCOMM_WORLD);
36      }
37
38       //-master get own data
39       sublist = splitCostModel(dataList,id,listOfChunks);// partitioning
40       //-master perform its task
41       if(masterInfo->numOfCores > 1){
42            //- call multi-core hReduce
43            reduce_result = MultiCorehReduce(sublist,listOfChunks[id],reduceFunc);
44            results[id] = reduce_result;
45       }
46       else{
47       //- call single-core hReduce
48           reduce_result = SingleCorehReduce(sublist,listOfChunks[id],reduceFunc);
49       results[id] = reduce_result;
50       }
51
52       //- master recieve data from the workers
53       for(dest=1;dest<np;dest++){
54           MPI_Recv(&results[dest], 1, ty, dest, 0, MPLCOMM_WORLD, &status);
55       }
56       /*
57        * master perform global reduce
58        * if it is only one machine( single value)
59       */
60        if(np == 1){
61              return results[0];
62        }
63        /*
64         * if the size of the data less than the number of cores
65         * then the skeleton uses multi core reduce
66        */
67         if(masterInfo->numOfCores<np){
68             skel_result = MultiCorehReduce(results, np,reduceFunc);
69         }
70         else{
71             skel_result = SingleCorehReduce(results,np,reduceFunc);
72         }
73         return skel_result;
74      }
75      else
76      {
77      //-Workers
```

```
78          MPI_Get_processor_name(name, &len);
79          ModelParas *workerInfo = GetProcInfo();
80
81          //-list of chunks
82          listOfChunks = (int*)malloc(np*sizeof(int));
83          MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
84
85        sublist = (void**) malloc(listOfChunks[id]*sizeof(void*));
86          MPI_Recv(sublist, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD, &status);
87          if(workerInfo->numOfCores > 1){
88               //- call multi-core hReduce
89               reduce_result = MultiCorehReduce(sublist, listOfChunks[id],
                     reduceFunc);
90               MPI_Send(&reduce_result, 1, ty, 0, 0, MPI_COMM_WORLD);
91          }
92          else{
93          //- call single-core hReduce
94               reduce_result = SingleCorehReduce(sublist,listOfChunks[id],
                    reduceFunc);
95               MPI_Send(&reduce_result, 1, ty, 0, 0, MPI_COMM_WORLD);
96         }
97     }
98 }
```

Listing A.7: *hReduce* Skeleton Code.

A.3.2 *hReduce* Single-Core

```
1
2 //- hReduce SingleCore
3
4 void* SingleCorehReduce(void* dataList,int size, void* funcName)
5 {
6   void *result;
7   //- casting the function pointer
8   void_fp = (void_pFun)funcName;
9   result = (void*)(*void_fp)(dataList,size);
10  return result;
11 }
```

Listing A.8: *hReduce SingleCore* Skeleton Code.

A.3.3 *hReduce* Multi-Core

```
1
2 //- hReduce MultiCore
3
4 void* MultiCorehReduce(void* dataList,int size,void* reduceFunc)
5 {
6     void *sublist;
7     void *reduce_result;
8     void **results;
9     int tid;
```

```
10      int threads = omp_get_max_threads();
11      int chunkSize = size/threads;
12      int remainder = size%threads;
13      int master_chunk = chunkSize+remainder;
14      results = (void**) malloc (threads*sizeof(void*));
15
16      //-parallel region
17      #pragma omp parallel private(tid,sublist,reduce_result) shared(dataList,
            results,chunkSize,master_chunk)
18      {
19       tid = omp_get_thread_num();
20         if(tid == 0){
21         sublist = splitEqual(dataList, master_chunk, remainder, tid);//-
               partitioning
22         //-local reduce
23               reduce_result = SingleCore(sublist, master_chunk,reduceFunc);
24          }
25          else{
26           sublist = splitEqual(dataList, chunkSize, remainder, tid);//-
                partitioning
27           //-local reduce
28         reduce_result = SingleCore(sublist, chunkSize,reduceFunc);
29      }
30          #pragma omp critical(update_Results)
31          {
32         results[tid] = reduce_result;
33          }
34       }
35       //-global reduce
36       reduce_result = SingleCore(results, threads,reduceFunc);
37       return reduce_result;
38  }
```

Listing A.9: *hReduce MultiCore* Skeleton Code.

A.4 *hMapReduce Skeleton*

This section presents the *hMapReduce* function discussed in Section 3.2.6.

A.4.1 *hMapReduce*

```
1
2 #include "HwSkel.h"
3
4 //- hMapReduce Skeleton
5
6 void* hMapReduce(void* dataList,int size,enum DataType dType, void* mapFunc,void
      * reduceFunc)
7 {
8   int i;              // index
9   int* listOfChunks; // list of chunks
10
11   void *sublist;
12     void *map_result;
```

```
13      void *reduce_result;
14      void *skel_result;
15
16      //- MPI_Datatype Conversion
17      ty = dataTypeConversion(dType);
18
19      if (StartNode)
20      {
21      printf("  MapReduce Skeleton \n\n");
22        int dest;
23          void **results;
24
25          results = (void**) malloc(np*sizeof(void*));
26          //-calculate the chunks size using cost model
27          listOfChunks = HybridCostModel(currentCluster,size);
28          MPI_Bcast(listOfChunks,np,MPI_INT,0,MPI_COMM_WORLD);
29
30          //-get master's specifications
31          ModelParas *masterInfo = ArrayList_GetItem(currentCluster,id);
32          //if(masterInfo->procName[0]=='l') masterInfo->numOfCores=1;
33
34          //- distribute the data
35          for(dest=1;dest<np;dest++){
36          sublist = splitCostModel(dataList,dest,listOfChunks);// partitioning
37              MPI_Send(sublist, listOfChunks[dest], ty, dest, 0, MPI_COMM_WORLD);
38          }
39
40          //- master get own data
41          sublist = splitCostModel(dataList,id,listOfChunks);// partitioning
42
43          //-master perform its task
44          //-check for multi-core system
45          if(masterInfo->numOfCores>1){
46            //- call multi-core hMapReduce
47                results[id] = MultiCorehMapReduce(sublist,listOfChunks[id],mapFunc
                        ,reduceFunc);
48          }
49              else{
50           //- call single-core hMapReduce
51                  map_result = SingleCorehMapReduce(sublist,listOfChunks[id],
                            mapFunc);
52                  reduce_result = SingleCorehMapReduce(map_result,listOfChunks[id
                            ],reduceFunc);
53                  results[id] = reduce_result;
54              }
55
56              //-master recieve data from the workers
57              for(dest=1;dest<np;dest++){
58            MPI_Recv(&results[dest], 1, ty, dest, 0, MPI_COMM_WORLD, &status);
59              }
60              /*
61               * master perform global reduce
62               * if it is only one machine( single value)
63              */
64              if(np == 1){
65                 return results[0];
66              }
67              /*
68               * if the size of the data less than the number of cores
69               * then the skeleton uses multi core reduce
70              */
71              if(masterInfo->numOfCores<np){  // check for multi-core system
72                  skel_result = MultiCoreReduce(results, np,reduceFunc);
73              }
74              else{
```

```
75                  skel_result = SingleCorehMapReduce(results,np,reduceFunc);
76              }
77              return skel_result;
78        }
79     }
80     else //----- Workers -----//
81     {
82         ModelParas *workerInfo = GetProcInfo();
83
84         //- list of chunks
85         listOfChunks = malloc (np*sizeof(int));
86         MPI_Bcast(listOfChunks,np,MPI_INT,0,MPI_COMM_WORLD);
87
88     //- worker recieve its portion of data
89         sublist = (void**) malloc(listOfChunks[id]*sizeof(void*));
90         MPI_Recv(sublist, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD, &status);
91
92         if(workerInfo->numOfCores>1){
93          //- call multi-core hMapReduce
94              skel_result = MultiCorehMapReduce(sublist, listOfChunks[id],
                    mapFunc,reduceFunc);
95              MPI_Send(&skel_result, 1, ty, 0, 0, MPI_COMM_WORLD);
96         }
97         else{
98              //- call single-core hMapReduce
99              map_result = SingleCorehMapReduce(sublist,listOfChunks[id],mapFunc)
                    ;
100             reduce_result = SingleCorehMapReduce(map_result,listOfChunks[id],
                    reduceFunc);
101             MPI_Send(&reduce_result, 1, ty, 0, 0, MPI_COMM_WORLD);
102         }
103   }
104 }
```

Listing A.10: *hMapReduce* Skeleton Code.

A.4.2 *hMapReduce* Single-Core

```
1
2 //- hMapReduce SingleCore
3
4 void* SingleCorehMapReduce(void* dataList,int size, void* funcName)
5 {
6   void *result;
7   void_fp = (void_pFun)funcName; //-casting the function pointer
8   result = (void*)(*void_fp)(dataList,size);
9   return result;
10 }
```

Listing A.11: *hMapReduce SingleCore* Skeleton Code.

A.4.3 *hMapReduce* Multi-Core

```
1  //- hMapReduce MultiCore
2
3  void* MultiCorehMapReduce(void* dataList,int size,void* mapFunc,void* reduceFunc
       )
4  {
5    void *sublist;
6      void *map_result;
7      void *reduce_result;
8      void **results;
9      int tid;
10     int threads = omp_get_max_threads();
11     int chunkSize = size/threads;
12     int remainder = size%threads;
13     int master_chunk = chunkSize+remainder;is shown
14     results = (void**) malloc (threads*sizeof(void*));
15
16     //-parallel region
17     #pragma omp parallel private(tid,sublist,map_result, reduce_result) shared(
           threads,dataList,results,chunkSize,master_chunk)
18     {
19       tid = omp_get_thread_num();
20         if(tid == 0){
21          sublist = splitEqual(dataList, master_chunk, remainder, tid);//-
                partitioning
22            map_result = SingleCore(sublist,master_chunk,mapFunc);
23            //-local reduce
24            reduce_result = SingleCore(map_result, master_chunk,reduceFunc);
25         }
26         else{
27        sublist = splitEqual(dataList, chunkSize, remainder, tid);//-partitioning
28        map_result = SingleCore(sublist, chunkSize,mapFunc);
29        //-local reduce
30        reduce_result = SingleCore(map_result, chunkSize,reduceFunc);
31     }
32         #pragma omp critical(update_Results)
33         {
34        results[tid] = reduce_result;
35         }
36      }
37      //-global reduce
38      reduce_result = SingleCore(results, threads,reduceFunc);
39      return reduce_result;
40 }
```

Listing A.12: *hMapReduce MultiCore* Skeleton Code.

Appendix B

The CM1 Cost Model

This appendix presents the complete code for the CM1 cost model discussed in Section 4.4, and also presents the code of *getClusterInfo()* (Section B.2) and *getNodeInfo()* (Section B.3) that collect and register the architectural information for the underlying hardware as discussed in Section 3.2.2.1.

B.1 The CM1 Code

```
1
2  //- CM1 Cost Model
3
4  int* CM1_CostModel(ArrayList* myParas, int dataSize)
5  {
6     //- struct of node information
7     ModelParas* paras;
8     //- list of chunks
9     int *listOfPortions;
10    //- total system power
11    float p=0.0;
12    int intpart=0;
13    int remainder;
14    int i;
15    for(i=0;i<np;i++)
16    {
17       paras = ArrayList_GetItem(myParas,i);
18       p = p + paras->numOfCores * paras->freq * paras->cacheSize;
19    }
20
21    //- calculate the chunk size
22    listOfPortions = malloc(np*sizeof(int));
23    for(i=0;i<np;i++)
```

```
24    {
25      //- apply CM1 cost model
26      paras = ArrayList_GetItem(myParas,i);
27        if(paras->numOfCores > 1){
28           listOfPortions[i] = paras->numOfCores * paras->freq * paras->cacheSize
                  / p * dataSize;
29           intpart+=listOfPortions[i];
30        }
31        else{
32           listOfPortions[i] = paras->freq * paras->cacheSize / p * dataSize;
33           intpart+=listOfPortions[i];
34        }
35    }
36
37    //- distribute the remainder between the nodes
38    remainder = dataSize - intpart;
39    int dest=0;
40    for(i=remainder;i>0;i--){
41       listOfPortions[dest]++;
42       dest++;
43       if(dest>=np) dest=0;
44    }
45
46    //- return the list of chunks
47    return listOfPortions;
48 }
```

Listing B.1: The CM1 cost model.

B.2 *getNodeInfo()*

```
1
2  //- Get Node Specifications
3
4  ModelParas* getNodeInfo()
5  {
6    char buff[128];
7      FILE * fin;
8      char *freq;
9      char *cacheSize;
10     if((fin = fopen("/proc/cpuinfo","r"))==NULL){
11     printf("Can not open the cpuinfo file !\n");
12         exit(EXIT_FAILURE);
13     }
14     ModelParas *procInfo = (ModelParas*)malloc(sizeof(ModelParas));
15     freq = malloc(8*sizeof(char));
16     cacheSize = malloc(4*sizeof(char));
17     while(fgets(buff, sizeof(buff), fin) != NULL)
18     {
19        if(strncmp(buff,"processor",9) == 0){
20           procInfo->numOfCores++;
21        }
22        else if(strncmp(buff,"cpu MHz",7) == 0){
23           strncpy(freq,buff+11,8);
24        }
25        else if(strncmp(buff,"cache size",10) == 0){
26           strncpy(cacheSize,buff+13,4);
27        }
```

```
28      }
29      fclose(fin);
30      procInfo->freq = atof(freq) / 1024.00;
31      procInfo->cacheSize = atof(cacheSize) / 1024.00;
32      return procInfo;
33  }
```

Listing B.2: Function for get node specifications.

B.3 getClusterInfo()

```
1
2   //-  Get Cluster Specifications
3
4   ArrayList* getClusterInfo()
5   {
6       int position;
7       char *buff;
8       unsigned int buffSize;
9       ModelParas *cpuInfo;
10      //- create an arraylist for cluster Specifications
11      ArrayList* clusterInfo = ArrayList_new0(NULL);
12      buffSize=2*sizeof(int)+2*sizeof(float)+MPI_MAX_PROCESSOR_NAME*sizeof(char);
13      buff=malloc(buffSize*sizeof(char));
14      if(StartNode)
15      {
16          MPI_Get_processor_name(name, &len);
17          //- get master specifications
18          cpuInfo=GetNodecInfo();
19          cpuInfo->procID = id;
20          memcpy(cpuInfo->procName,name,strlen(name)*sizeof(char));
21          ArrayList_AddItem(clusterInfo,cpuInfo);
22          int dest;
23          for(dest=1;dest<np;dest++)
24          {
25            //- master recieve node specifications
26            MPI_Recv(buff, buffSize, MPI_PACKED,dest,0, MPI_COMM_WORLD, &status);
27            cpuInfo = ModelParas_new();
28            position=0;
29            MPI_Unpack(buff, buffSize, &position,&cpuInfo->procName,
                  MPI_MAX_PROCESSOR_NAME, MPI_CHAR, MPI_COMM_WORLD);
30            MPI_Unpack(buff, buffSize, &position,&cpuInfo->procID,1, MPI_INT,
                  MPI_COMM_WORLD);
31            MPI_Unpack(buff, buffSize, &position,&cpuInfo->numOfCores,1, MPI_INT,
                  MPI_COMM_WORLD);
32            MPI_Unpack(buff, buffSize, &position,&cpuInfo->freq,1, MPI_FLOAT,
                  MPI_COMM_WORLD);
33            MPI_Unpack(buff, buffSize, &position,&cpuInfo->cacheSize,1, MPI_FLOAT,
                   MPI_COMM_WORLD);
34            ArrayList_AddItem(clusterInfo,cpuInfo);
35        }
36      }
37      else
38      {
39         MPI_Get_processor_name(name, &len);
40         //- get worker's specifications
41         cpuInfo=GetNodeInfo();
42         //- worker send the specifications
```

```
43          position = 0;
44          MPI_Pack(name,MPI_MAX_PROCESSOR_NAME, MPI_CHAR,buff, buffSize, &position,
                MPI_COMM_WORLD);
45          MPI_Pack(&id,1, MPI_INT,buff, buffSize, &position, MPI_COMM_WORLD);
46          MPI_Pack(&cpuInfo->numOfCores,1, MPI_INT,buff, buffSize, &position,
                MPI_COMM_WORLD);
47          MPI_Pack(&cpuInfo->freq,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
48          MPI_Pack(&cpuInfo->cacheSize,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
49          MPI_Send(buff,buffSize,MPI_PACKED,0,0,MPI_COMM_WORLD);
50        }
51        return clusterInfo;
52  }
```

Listing B.3: Function for get cluster specifications.

Appendix C

The *GPU-HWSkel* Library

This appendix presents the *GPU-HWSkel* library. All the skeletons functions in *GPU-HWSkel* library are similar to those presented in the *HWSkel* library, the only difference is that multicore and single core functions are modified to handle GPU device. Thus, in this appendix we present the *hMap* to show how the GPU is being controlled by one of the cores in the system.

C.1 *hMap Skeleton*

This section presents the *hMap* function and show how to use CM2 cost model (line 26 of C.1) presented in Appendix D.

C.1.1 *hMap*

```
1
2  //- hMap Function
3
4  void* hMap(void* dataList, int size, enum DataType dType)
5  {
6      int i;              // index
7      int* listOfChunks; // list of chunks
8
9      //- UNSERIALIZED variables
10     void *sublist;
```

```
11      void *map_result;
12      void **skel_result;
13
14      //- MPI_Datatype Conversion
15      ty = dataTypeConversion(dType);
16
17      if (StartNode)
18      {
19          printf("\nCall heterogeneous map skeleton !\n");
20          int dest;
21      int offset;
22
23      skel_result = (void**) malloc(size*sizeof(void*));
24
25      //- Call  multi-node CM2 cost model
26      listOfChunks = ClusterCostModel(currentCluster,size);
27
28
29      MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
30
31      //-get master's specifications
32      ModelParas *masterInfo = ArrayList_GetItem(currentCluster,id);
33
34      //-master split and send data to workers
35      for(dest=1;dest<np;dest++){
36        sublist = splitCostModel(dataList,dest,listOfChunks);// partitioning
37        MPI_Send(sublist, listOfChunks[dest], ty, dest, 0, MPI_COMM_WORLD);
38      }
39
40      //-master get own data
41      sublist = splitCostModel(dataList,id,listOfChunks);// partitioning
42
43      //- check for multi-core system
44      if(masterInfo->cpuCores > 1){
45        printf(" Master --> multi-core machine  \n");
46        map_result = MultiCoreMap(sublist,listOfChunks[id], masterInfo);
47        offset = listOfChunks[id];
48        }
49      else{
50              printf("  Master --> single-core machine \n");
51        map_result = SingleCoreMap(sublist, listOfChunks[id], masterInfo->
              gpuDevices);
52        offset = listOfChunks[id];
53         }
54          memcpy( skel_result,map_result,listOfChunks[id]*sizeof(void*));
55
56
57      //-master recieve data from the workers
58      for(dest=1;dest<np;dest++){
59          printf("worker(%d) len = %d\n", dest, listOfChunks[dest]);
60        MPI_Recv(map_result, listOfChunks[dest], ty, dest, 0, MPI_COMM_WORLD, &
              status);
61        memcpy( skel_result+offset,map_result,listOfChunks[dest]*sizeof(void*));
62        offset+=listOfChunks[dest];
63      }
64          return skel_result;
65
66    }
67      else
68      {
69      //- Workers
70        MPI_Get_processor_name(name, &len);
71          ModelParas *workerInfo = GetNodeInfo();
72
73          //-list of chunks
```

```
74          listOfChunks = (int*)malloc(np*sizeof(int));
75          MPI_Bcast(listOfChunks, np, MPI_INT, 0, MPI_COMM_WORLD);
76
77        sublist = (void**) malloc(listOfChunks[id]*sizeof(void*));
78          MPI_Recv(sublist, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD, &status);
79
80          if(workerInfo->cpuCores > 1){
81        printf(" Worker --> multi-core machine  \n");
82              map_result = MultiCoreMap(sublist, listOfChunks[id], workerInfo);
83          }
84          else{
85              printf("  worker --> single-core machine \n");
86              map_result = SingleCoreMap(sublist,listOfChunks[id], workerInfo->
                    gpuDevices);
87          }
88          MPI_Send(map_result, listOfChunks[id], ty, 0, 0, MPI_COMM_WORLD);
89          return 0;
90
91      }
92  }
```

Listing C.1: *hMap* Skeleton Code.

C.1.2 *hMap* Single-Core

```
1
2
3  //- SingleCore hMap
4
5  void* SingleCore(void* dataList, int size, int cudaVersionFlag)
6  {
7    void **results = (void**)malloc(size*sizeof(void*));
8
9    if(cudaVersionFlag != 1){
10      cpu_map(dataList, size);
11   }
12   else{
13      cuda_map(dataList, size);
14   }
15
16   memcpy(results,dataList,size*sizeof(void*));
17
18  return results;
19 }
```

Listing C.2: *hMap SingleCore* Skeleton Code.

C.1.3 *hMap* Multi-Core

```
1
2  //- MultiCore hMap
3
4  void* MultiCore(void* dataList,int size, ModelParas* nodeInfo)
```

```
5  {
6    int i;
7      void *sublist;
8      void **results;
9      int offset;
10     int tid;
11
12     //- Call single-node CM2 cost model
13     int *listOfChunks = NodeCostModel(size, nodeInfo);
14     results = (void**)malloc(size*sizeof(void*));
15
16     //- parallel region
17   #pragma omp parallel private(tid,sublist,offset) shared(dataList,results,
         listOfChunks)
18     {
19       tid = omp_get_thread_num();
20       if(tid == 0){
21         //- partitioning
22         sublist = splitCostModel(dataList, tid, listOfChunks);
23         //- controlling GPU device
24         SingleCoreMap(sublist, listOfChunks[tid], nodeInfo->gpuDevices);
25       }
26       else{
27         //- partitioning
28         sublist = splitCostModel(dataList, tid, listOfChunks);
29         SingleCoreMap(sublist, listOfChunks[tid], 0);
30       }
31       #pragma omp critical(update_Results)
32       {
33         offset = 0;
34         for(i=0; i<tid; i++) offset+=listOfChunks[i];
35         memcpy(results+offset,sublist,listOfChunks[tid]*sizeof(void*));
36       }
37     }
38     return results;
39 }
```

Listing C.3: *hMapMultiCore* Skeleton Code.

Appendix D

The CM2 Cost Model

This appendix presents the complete code for the CM2 cost model discussed in Section 6.3.

D.1 *getNodeInfo()*

```
1
2
3  //- Get Node Specifications
4
5  ModelParas* GetNodeInfo()
6  {
7     char buff[128];
8     FILE * fin;
9     char *freq;
10    float speed;
11    char *cache;
12    float cacheSize;
13    if((fin = fopen("/proc/cpuinfo","r"))==NULL){
14       printf("Can not open the cpuinfo file !\n");
15         exit(EXIT_FAILURE);
16    }
17    ModelParas *nodeInfo=(ModelParas*)malloc(sizeof(ModelParas));
18     freq=malloc(8*sizeof(char));
19     cache=malloc(4*sizeof(char));
20     while(fgets(buff, sizeof(buff), fin) != NULL)
21     {
22        if(strncmp(buff,"processor",9) == 0){
23           nodeInfo->cpuCores++;
24        }
25        else if(strncmp(buff,"cpu MHz",7) == 0){
26           strncpy(freq,buff+11,8);
27        }
28        else if(strncmp(buff,"cache size",10) == 0){
```

```
29              strncpy(cache,buff+13,4);
30          }
31      }
32      fclose(fin);
33      speed = atof(freq);
34      nodeInfo->cpuFreq = speed / 1024.00;;
35      cacheSize = atof(cache);
36      nodeInfo->cpuCacheSize = cacheSize / 1024.00;
37
38      //- GPU information
39      nodeInfo->gpuDevices = checkCudaDevices();
40      if(nodeInfo->gpuDevices != 0){
41      nodeInfo->gpuCores = getGpuCores();
42          nodeInfo->gpuFreq = getGpuFreq() / 1024; //MHz
43          nodeInfo->gpuCacheSize = getGpuCacheSize() / 1024; //MB
44          //- get Gpu time
45          nodeInfo->gpuTime = getGpuTime();
46      }
47
48      //- get Cpu sequential time
49      nodeInfo->cpuSeqTime = getCpuSeqTime();
50
51
52      return nodeInfo;
53  }
```

Listing D.1: Node Information.

D.2 getClusterInfo()

```
1
2
3   //- Get cluster information
4
5   ArrayList* GetClusterInfo()
6   {
7       int position;
8       char *buff;
9       int gpuDevices;
10      int gpuCores;
11      unsigned int buffSize;
12      ModelParas *pcInfo;
13
14      ArrayList* clusterInfo = ArrayList_new0(NULL);
15      buffSize=4*sizeof(int)+6*sizeof(float)+MPI_MAX_PROCESSOR_NAME*sizeof(char);
16      buff=malloc(buffSize*sizeof(char));
17
18      if(StartNode)
19      {
20          //- get node's name
21          MPI_Get_processor_name(name, &len);
22
23          //- get master node information
24          pcInfo = GetNodeInfo();
25          pcInfo->procID = id;
26
27          memcpy(pcInfo->procName,name,strlen(name)*sizeof(char));
28          ArrayList_AddItem(clusterInfo,pcInfo);
```

```
29
30          int dest;
31          for(dest=1;dest<np;dest++)
32          {
33              MPI_Recv(buff, buffSize, MPI_PACKED,dest,0, MPI_COMM_WORLD, &status);
34              pcInfo = ModelParas_new();
35              position=0;
36              MPI_Unpack(buff, buffSize, &position, &pcInfo->procName,
                    MPI_MAX_PROCESSOR_NAME, MPI_CHAR, MPI_COMM_WORLD);
37              MPI_Unpack(buff, buffSize, &position, &pcInfo->procID, 1, MPI_INT,
                    MPI_COMM_WORLD);
38              MPI_Unpack(buff, buffSize, &position, &pcInfo->cpuCores, 1, MPI_INT,
                    MPI_COMM_WORLD);
39              MPI_Unpack(buff, buffSize, &position, &pcInfo->cpuFreq, 1, MPI_FLOAT,
                     MPI_COMM_WORLD);
40              MPI_Unpack(buff, buffSize, &position, &pcInfo->cpuCacheSize, 1,
                    MPI_FLOAT, MPI_COMM_WORLD);
41              MPI_Unpack(buff, buffSize, &position, &pcInfo->gpuDevices, 1, MPI_INT
                    , MPI_COMM_WORLD);
42              MPI_Unpack(buff, buffSize, &position, &pcInfo->gpuCores, 1, MPI_INT,
                    MPI_COMM_WORLD);
43              MPI_Unpack(buff, buffSize, &position, &pcInfo->gpuFreq, 1, MPI_FLOAT,
                     MPI_COMM_WORLD);
44              MPI_Unpack(buff, buffSize, &position, &pcInfo->gpuCacheSize, 1,
                    MPI_FLOAT, MPI_COMM_WORLD);
45              MPI_Unpack(buff, buffSize, &position, &pcInfo->cpuSeqTime, 1,
                    MPI_FLOAT, MPI_COMM_WORLD)
46              MPI_Unpack(buff, buffSize, &position, &pcInfo->gpuTime, 1, MPI_FLOAT,
                     MPI_COMM_WORLD);
47              ArrayList_AddItem(clusterInfo,pcInfo);
48          }
49      }
50      else
51      {
52          //- get node's name
53          MPI_Get_processor_name(name, &len);
54
55          //- get worker node information
56          pcInfo = GetNodeInfo();
57
58          //- worker send the information
59          position = 0;
60          MPI_Pack(name,MPI_MAX_PROCESSOR_NAME, MPI_CHAR,buff, buffSize, &position
                , MPI_COMM_WORLD);
61          MPI_Pack(&id,1, MPI_INT,buff, buffSize, &position, MPI_COMM_WORLD);
62          MPI_Pack(&pcInfo->cpuCores,1, MPI_INT,buff, buffSize, &position,
                MPI_COMM_WORLD);
63          MPI_Pack(&pcInfo->cpuFreq,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
64          MPI_Pack(&pcInfo->cpuCacheSize,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
65          MPI_Pack(&pcInfo->gpuDevices, 1, MPI_INT, buff, buffSize, &position,
                MPI_COMM_WORLD);
66          MPI_Pack(&pcInfo->gpuCores, 1, MPI_INT, buff, buffSize, &position,
                MPI_COMM_WORLD);
67          MPI_Pack(&pcInfo->gpuFreq,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
68          MPI_Pack(&pcInfo->gpuCacheSize,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
69          MPI_Pack(&pcInfo->cpuSeqTime,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
70          MPI_Pack(&pcInfo->gpuTime,1, MPI_FLOAT,buff, buffSize, &position,
                MPI_COMM_WORLD);
71          MPI_Send(buff,buffSize,MPI_PACKED,0,0,MPI_COMM_WORLD);
72      }
```

```
73      return clusterInfo;
74  }
```

Listing D.2: Cluster Information.

D.3 GPU Information

```
1
2
3  //-check for Cuda Device
4
5  extern "C" int checkCudaDevices()
6  {
7     int deviceCount = 0;
8     cudaGetDeviceCount(&deviceCount);
9     return deviceCount;
10 }
11
12 //- get gpu clock rate
13
14 extern "C" int getGpuFreq()
15 {
16    cudaDeviceProp devProp;
17    int clockRate;
18    int deviceCount = 0;
19    cudaGetDeviceCount(&deviceCount);
20    if( deviceCount != 0 ){
21       cudaGetDeviceProperties(&devProp, 0);
22       clockRate = devProp.clockRate;
23       return clockRate;
24    }
25    return 0;
26 }
27
28 //- get gpu l2 cache size
29
30  extern "C" int getGpuCacheSize()
31  {
32    cudaDeviceProp devProp;
33    int cacheSize;
34    int deviceCount = 0;
35    cudaGetDeviceCount(&deviceCount);
36    if( deviceCount != 0 ){
37       cudaGetDeviceProperties(&devProp, 0);
38       cacheSize = devProp.l2CacheSize;
39       return cacheSize;
40    }
41    return 0;
42 }
43
44 //- Get GPU cores
45
46 extern "C" int getGpuCores()
47 {
48    cudaDeviceProp devProp;
49    int cudaMultiprocessor;
50    int cudaCores;
51    int deviceCount = 0;
```

```
52     cudaGetDeviceCount(&deviceCount);
53     if( deviceCount != 0 ){
54        cudaGetDeviceProperties(&devProp, 0);
55        cudaMultiprocessor = devProp.multiProcessorCount;
56        cudaCores = cudaMultiprocessor;
57        return cudaCores;
58     }
59     return 0;
60 }
```

Listing D.3: GPU Information.

D.4 Single-Node Cost Model

```
1
2  //- Node-level Cost Model
3
4  int *NodeCostModel(int size, ModelParas *nodeParas)
5  {
6     int i;
7     int cpuChunk, gpuChunk, coreChunk;
8     int remainder;
9     int chunkSize;
10    float ratio;
11
12    int *chunksList = malloc(nodeParas->cpuCores*sizeof(int));
13
14    if(nodeParas->gpuDevices == 0){
15        remainder = size%nodeParas->cpuCores;
16        chunkSize = size/nodeParas->cpuCores;
17        chunksList[0] = chunkSize+remainder;
18        for(i=1;i<nodeParas->cpuCores;i++)
19             chunksList[i]= chunkSize;
20    }
21    else{
22        ratio = nodeParas->cpuSeqTime / nodeParas->gpuTime;
23        cpuChunk = size / (1 + ratio /(nodeParas->cpuCores -1));
24        gpuChunk = size - cpuChunk;
25
26        chunksList[0] = gpuChunk;
27        chunkSize = cpuChunk / (nodeParas->cpuCores -1);
28        remainder = cpuChunk % (nodeParas->cpuCores - 1);
29        chunksList[1] = chunkSize+remainder;
30        for(i=2;i<nodeParas->cpuCores;i++)
31            chunksList[i] = chunkSize;
32    }
33    return chunksList;
34 }
```

Listing D.4: Function for get node specifications.

D.5 Multi-Node Cost Model

```
1
2
3  //- Multi-Node CM2 Cost Model
4
5  int* ClusterCostModel(ArrayList* myParas, int dataSize)
6  {
7     int i;
8     ModelParas* paras;
9     int *listOfPortions = malloc(np*sizeof(int));
10    float total = 0.0;
11    int intpart = 0;
12    int remainder;
13    float hardwareBase;
14    float nodePower;
15
16    paras = ArrayList_GetItem(myParas,0);
17    hardwareBase = paras->cpuFreq * paras->cpuCacheSize * paras->cpuCores;
18
19    //- calaculate total processing power
20    for(i = 0; i < np; i++)
21    {
22      paras = ArrayList_GetItem(myParas,i);
23      if(paras->gpuDevices == 0){
24         nodePower = paras->cpuCores * (paras->cpuFreq*paras->cpuCacheSize*paras
              ->cpuCores) / hardwareBase;
25         total += nodePower;
26      }
27      else{
28          nodePower = ((paras->cpuCores -1) * (paras->cpuFreq*paras->cpuCacheSize
               *paras->cpuCores) / hardwareBase)
29                    + (tBase / paras->gpuTime);
30          total += nodePower;
31      }
32    }
33
34    //- calaculate the chunk size
35    for(i = 0; i < np; i++)
36    {
37      paras = ArrayList_GetItem(myParas,i);
38      if(paras->gpuDevices == 0){
39         nodePower = paras->cpuCores *  (paras->cpuFreq*paras->cpuCacheSize*paras
              ->cpuCores) / hardwareBase;
40         listOfPortions[i] = nodePower / total * dataSize;
41         intpart += listOfPortions[i];
42      }
43      else{
44         nodePower = ((paras->cpuCores -1) * (paras->cpuFreq*paras->cpuCacheSize*
              paras->cpuCores)) / hardwareBase + (tBase / paras->gpuTime);
45         listOfPortions[i] = nodePower / total * dataSize;
46         intpart += listOfPortions[i];
47      }
48    }
49
50    //- distribute the remainder between the workers
51    remainder = dataSize - intpart;
52    listOfPortions[0] += remainder;
53
54    return listOfPortions;
55 }
```

Listing D.5: Function for get cluster specifications.

Bibliography

[1] NVIDIA. CUDA Programming Guide, Version 4.2. Technical report, 2012.

[2] E. Wu and Y. Liu. Emerging Technology about GPGPU. In *APCCAS 2008. IEEE Asia Pacific Conference on Circuits and Systems, 2008.*, pages 618–622, 2008.

[3] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[4] G. Jost, H. Jin, D. an Mey, and F. F. Hatay. Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster. EWOMP'03, Aachen, Germany, 2003.

[5] E. Lusk and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In *IWOMP'08: Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, pages 36–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, pages 427–436, 2009.

[7] L. A. Smith. Mixed mode MPI/OpenMP programming. *UK High-End Computing Technology Report*, 2000.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[9] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. In *SIGARCH Comput. Archit. News*, volume 34, pages 325–335, New York, NY, USA, October 2006. ACM.

[10] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.

[11] C. Luk, S. Hong, and H. Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[12] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, MA, 1989.

[13] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 1999.

[14] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, UK, 2003.

[15] M. Leyton. *Advanced Features for Algorithmic Skeleton Programming*. PhD thesis, Universite de Nice - Sophia Antipolis – UFR Sciences, October 2008.

[16] P. Ciechanowicz and H. Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, pages 108–113, Washington, DC, USA, 2010. IEEE Computer Society.

[17] Y. Karasawa and H. Iwasaki. A Parallel Skeleton Library for Multi-core Clusters. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 84–91, Washington, DC, USA, 2009. IEEE Computer Society.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation)*. The MIT Press, November 1999.

[19] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, October 2007.

[20] K. Armih, G. Michaelson, and P. Trinder. Cache Size in a Cost Model for Heterogeneous Skeletons. In *Proceedings of the Fifth International Workshop on High-level Parallel Programming and Applications*, HLPP '11, pages 3–10, New York, NY, USA, 2011. ACM.

[21] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[22] H. El-Rewini. *Advanced Computer Architecture and Parallel Processing; electronic version*. Wiley, New York, NY, 2004.

[23] M. J. Flynn. Some Computer Organizations and Their Effectiveness. In *IEEE Trans. Comput.*, volume 21, pages 948–960, Washington, DC, USA, September 1972. IEEE Computer Society.

[24] G. R. Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[25] T. Sterling, D. J. Becker, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A Parallel Workstation For Scientific Computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14, Urbana-Champain, Illinois, USA, 1995. CRC Press.

[26] P. A. Revenga, J. Sérot, J. L. Lázaro, and J. P. Derutin. A Beowulf-Class Architecture Proposal for Real-Time Embedded Vision. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 232.2, Washington, DC, USA, 2003. IEEE Computer Society.

[27] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT, 1994.

[28] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[29] S. Akhter and J. Roberts. *Multi-Core Programming: Increasing Performance through Software Multi-threading*. Richard Bowles, 2006.

[30] Intel Core2 Duo Processor. http://www.intel.com/products/processor/core2duo/.

[31] Athlon 64 X2 . http://www.amd.com/us/products/desktop/processors/athlon-x2/Pages/athlon-x2-faqs-black-edition.aspx.

[32] Intel Xeon Processor E5410. http://ark.intel.com/products/33080/.

[33] Intel Xeon 'Nehalem-EX' Processor. http://www.intel.com/pressroom/archive/-releases/2009/20090526comp.htm.

[34] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC Processor: The Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[35] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[36] K. S. Perumalla. Discrete-event Execution Alternatives on General Purpose Graphical Processing Units (GPGPUs). In *PADS '06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 74–81, Washington, DC, USA, 2006. IEEE Computer Society.

[37] NVIDIA CUDA Compute Unified Device Architecture: Programming guide v2.0. 2008.

[38] High Performance Computing - Supercomputing with Tesla GPUs. http://www.nvidia.com/object/tesla-supercomputing-solutions.html.

[39] T. Sato. The Earth Simulator: Roles and Impacts. *Parallel Comput.*, 30(12):1279–1286, December 2004.

[40] E. Strohmaier and J. D. Meuer. TOP500 Supercomputer Sites. Tech report, University of Tennessee, Knoxville, TN, USA, 1997.

[41] R. Buyya. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[42] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[43] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[44] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Concurrency*, 4(2):63–79, 1996.

[45] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, 1991.

[46] S. Gorlatch, F. A. Rabhi, and S. Gorlatch. A Programming Methodology with Skeletons and Collective Operations. In *Patterns and Skeletons for Parallel and Distributed Computing*, pages 29–63. Springer-Verlag, London, UK, UK, 2003.

[47] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[48] R. F. Pointon, P. W. Trinder, and H-W. Loidl. The Design and Implementation of Glasgow Distributed Haskell. In *Selected Papers from the 12th*

International Workshop on Implementation of Functional Languages, IFL '00, pages 53–70, London, UK, UK, 2001. Springer-Verlag.

[49] M Aswad, P. W. Trinder, and H-W Loidl. Architecture Aware Parallel Programming in Glasgow Parallel Haskell (GPH). In *ICCS*, volume 9 of *Procedia Computer Science*, pages 1807–1816, Omaha, Nebraska, USA, 2012. Elsevier.

[50] Intel Threading Building Blocks . http://www.threadingbuildingblocks.org/.

[51] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating Code on Multi-cores with FastFlow. In *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, August 2011. Springer.

[52] S. Gupta. Performance Analysis of GPU Compared to Single-Core and Multi-Core CPU for Natural Language Applications. *IJACSA - International Journal of Advanced Computer Science and Applications*, 2(5):50–53, 2011.

[53] M. McCool and S. D. Toit. *Metaprogramming GPUs with Sh*. AK Peters Ltd, 2004.

[54] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1 edition, July 2011.

[55] AMD Corporation. ATI Stream Computing User Guide, Version 2.01. Technical report, 2010.

[56] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *The Fifth*

International Workshop on Automatic Performance Tuning, UC Berkeley - CITRIS, Sutardja Dai Hall, Berkeley, CA 94720, USA, June 2010.

[57] R. Rabenseifner. Hybrid Parallel Programming on HPC Platforms. *Proc. European Workshop on OpenMP '03, Aachen, Germany*, 2003.

[58] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs.* PhD thesis, University of Pisa, March 1993.

[59] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming.* Addison-Wesley Professional, 2007.

[60] A. J. Dorta, J. A. González, C. Rodríguez, and F. de Sande. llc: A Parallel Skeletal Language. *Parallel Processing Letters*, 13(3):437–448, September 2003.

[61] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M Vanneschi. A Methodology for the Development and the Support of Massively Parallel Programs. *Future Gener. Comput. Syst.*, 8(1-3):205–220, 1992.

[62] G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC5)*, pages 243–252. Society Press, 1996.

[63] C. A. Herrmann and C. Lengauer. HDC: A Higher-Order Language for Divide-and-Conquer. *Parallel Processing Letters*, 10(2/3):239–250, 2000.

[64] M. Cole. Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Comput.*, 30(3):389–406, 2004.

[65] H. Kuchen. A Skeleton Library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.

[66] M. Aldinucci, M. Danelutto, and P. Dazzi. MUSKEL: An Expandable Skeleton Environment. *Scientific International Journal for Parallel and Distributed Computing*, Vol. 8:325–341, December 2007.

[67] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for Data Parallelism in P3L. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 619–628, London, UK, 1997. Springer-Verlag.

[68] H. Kuchen and M. Cole. The Integration of Task and Data Parallel Skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.

[69] M. Danelutto and M. Aldinucci. Algorithmic Skeletons Meeting Grids. *Parallel Computing*, 32(7-8):449–462, 2006.

[70] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers. *Software–Practice & Experience*, 40(12):1135–1160, November 2010.

[71] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A Structured High-level Parallel Language, and its Structured Support. Technical report, Pisa Science Center, Hewlett-Packard Laboratory, 1995.

[72] G. H. Botorog and H. Kuchen. Using Algorithmic Skeletons with Dynamic Data Structures. In *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '96, pages 263–276, London, UK, 1996. Springer-Verlag.

[73] S. Pelagatti. Task and Data Parallelism in P3L. In Fethi A. Rabhi and Sergei Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, pages 155–186. Springer-Verlag, London, UK, 2003.

[74] H. Kuchen. Optimizing Sequences of Skeleton Calls. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 254–273. Springer Berlin Heidelberg, 2004.

[75] M. Aldinucci, M. Danelutto, and P. Teti. An advanced Environment Supporting Structured Parallel Programming in Java. *Future Gener. Comput. Syst.*, 19(5):611–626, July 2003.

[76] M. Aldinucci, M. Danelutto, and J. Dnnweber. Optimization Techniques for Implementing Parallel Skeletons in Grid Environments. In *CMPP04 Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, 2004.

[77] A. Benoit and M. Cole. Two Fundamental Concepts in Skeletal Parallel Programming. In *The International Conference on Computational Science (ICCS 2005) , Part II, LNCS 3515*, pages 764–771. Springer Verlag, 2005.

[78] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *11th Intl Euro-Par: Parallel and Distributed Computing, vol. 3648 of LNCS, 761-770, Lisbona*, pages 761–770. Springer-Verlag, 2005.

[79] M. Danelutto and P. Dazzi. Joint Structured/Unstructured Parallelism Exploitation in muskel. In V. Alexandrov, G. Albada, P. Sloot, and J. Dongarra, editors, *Computational Science ICCS 2006*, volume 3992 of *Lecture*

Notes in Computer Science, pages 937–944. Springer Berlin Heidelberg, 2006.

[80] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale '06: Proceedings of the 1st International Conference on Scalable Information Systems*, page 13, New York, NY, USA, 2006. ACM.

[81] R. Backhouse. An Exploration of the Bird-Meertens Formalism. Technical report, In STOP Summer School on Constructive Algorithmics, Abeland, September 1989.

[82] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A Fusion-Embedded Skeleton Library. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 644–653. Springer, 2004.

[83] K. Matsuzaki and K. Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *Draft Proceedings of the 21st International Symposium on Implementation and Application of Functional Languages*, pages 100–115, Seton Hall University, 2009.

[84] D. Caromel and M. Leyton. Fine Tuning Algorithmic Skeletons. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer Berlin Heidelberg, 2007.

[85] ProActive. http://proactive.activeeon.com/index.php.

[86] J. Falcou, J. Srot, T. Chateau, and J. T. Laprest. Quaff: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32(78):604 – 615, 2006.

[87] M. Leyton and J. M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, pages 289–296, 2010.

[88] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-level and Efficient Streaming on Multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, pages 1 – 20. Wiley, January 2013.

[89] M. Alind, M. V. Eriksson, and C. W. Kessler. BlockLib: A Skeleton Library for Cell Broadband Engine. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, IWMSE '08, pages 7–14, New York, NY, USA, 2008. ACM.

[90] C. W. Keler. NestStep: Nested Parallelism and Virtual Shared Memory for the BSP model. *The Journal of Supercomputing*, 17:613 – 619, 1999.

[91] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Skeletons for Multi/Many-core Systems. In B. Chapman, F. Desprez, G. R. Joubert, A. Lichnewsky, T. Priol, and F. J. Peters, editors, *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, volume 19 of *Advances in Parallel Computing*, pages 265–272, Lyon, France, September 2009. IOS press.

[92] A. Dorta, P. López, and F. de Sande. Basic Skeletons in llc. *Parallel Comput.*, 32(7):491–506, September 2006.

[93] R. Reyes, A. J. Dorta, F. Almeida, and F. Sande. Automatic Hybrid MPI+OpenMP Code Generation with llc. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel*

Virtual Machine and Message Passing Interface, pages 185–195, Berlin, Heidelberg, 2009. Springer-Verlag.

[94] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 225–234, New York, NY, USA, 2008. ACM.

[95] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009.

[96] J. Hoberock and N. Bell. Thrust: A Parallel Template Library. `http://code.google.com/p/thrust`, 2012.

[97] S. Sengupta Y. Zhang M. Harris, J. Owens and A. Davidson. CUDPP: CUDA Data Paralle Primitives Library. `https://code.google.com/p/cudpp/`, 2011.

[98] J. Enmyren and C. W. Kessler. SkePU: A Multi-backend Skeleton Programming Library for multi-GPU Systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[99] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, pages 1176–1182, Washington, DC, USA, 2011. IEEE Computer Society.

[100] U. Dastgeer, J. Enmyren, and C. W. Kessler. Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for multi-GPU Systems. In *Proceedings of the 4th International Workshop on Multicore Software Engineering*, IWMSE '11, pages 25–32, New York, NY, USA, 2011. ACM.

[101] S. Ernsting and H. Kuchen. Data Parallel Skeletons for GPU Clusters and Multi-GPU Systems. In *PARCO*, volume 22 of *Advances in Parallel Computing*, pages 509–518, Ghent, Belgium, 2011. IOS Press.

[102] S. Ernsting and H. Kuchen. Algorithmic Skeletons for Multi-core, Multi-GPU Systems and Clusters. *Int. J. High Perform. Comput. Netw.*, 7(2):129–138, April 2012.

[103] M. Goli, M. T. Garba, and H. González-Vélez. Streaming Dynamic Coarse-Grained CPU/GPU Workloads with Heterogeneous Pipelines in FastFlow. In *HPCC-ICESS*, pages 445–452, Liverpool, UK, 2012. IEEE Computer Society.

[104] A. Wikstrom. *Functional Programming Using Standard ML*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1987.

[105] H. Kuchen and J. Striegnitz. Higher-order Functions and Partial Applications for A C++ Skeleton Library. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, JGI '02, pages 122–130, New York, NY, USA, 2002. ACM.

[106] R. Bird and O. de Moor. *Algebra of programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[107] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[108] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, 1995.*, volume 2, pages 61–70 vol.2, 1995.

[109] S. E. Hambrusch. Models for Parallel Computation. In *ICPP Workshop*, pages 92–95, 1996.

[110] B. H. H. Juurlink and H. A. G. Wijshoff. A Quantitative Comparison of Parallel Computation Models. *ACM Trans. Comput. Syst.*, 16(3):271–318, August 1998.

[111] P. Trinder, M. Cole, K. Hammond, H-W. Loidl, and G. Michaelson. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience*, 25(3):309–348, 2013.

[112] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[113] S. A. Cook and R. A. Reckhow. Time-Bounded Random Access Machines. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, pages 73–80, New York, NY, USA, 1972. ACM.

[114] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computations. In *Proceedings of The First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 11–21, New York, NY, USA, 1989. ACM.

[115] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theor. Comput. Sci.*, 71(1):3–28, March 1990.

[116] C. Martel and A. Raghunathan. Asynchronous PRAMs with Memory Latency. *J. Parallel Distrib. Comput.*, 23(1):10–26, October 1994.

[117] R. Cole and O. Zajicek. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 169–178, New York, NY, USA, 1989. ACM.

[118] T. Heywood and S. Ranka. A Practical Hierarchical Model of Parallel Computation I. The Model. *Journal of Parallel and Distributed Computing*, 16(3):212 – 232, 1992.

[119] P. B. Gibbons. A More Practical PRAM Model. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 158–168, New York, NY, USA, 1989. ACM.

[120] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[121] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for Contention in Parallel Algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, pages 638–648, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[122] P. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *Proceedings of The Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 236–247, New York, NY, USA, 1994. ACM.

[123] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33:103–111, August 1990.

[124] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[125] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14):1947 – 1980, 1998.

[126] A. Zavanella. Skel-BSP: Performance Portability for Skeletal Programming. In *Proceedings of the 8th International Conference on High-Performance Computing and Networking*, HPCN Europe 2000, pages 290–299, London, UK, UK, 2000. Springer-Verlag.

[127] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 1–12, New York, NY, USA, 1996. ACM.

[128] A. Goldchleger, A. Goldman, U. Hayashida, and F. Kon. The implementation of the BSP Parallel Computing Model on the InteGrade Grid Middleware. In *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, MGC '05, pages 1–6, New York, NY, USA, 2005. ACM.

[129] P. de la Torre and C. Kruskal. Submachine Locality in the Bulk Synchronous Setting. In L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 352–358. Springer Berlin / Heidelberg, 1996.

[130] B. H. Juurlink and H. G. Wijshoff. The E-BSP model: Incorporating General Locality and Unbalanced Communication into the BSP Model. In L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 339–347. Springer Berlin Heidelberg, 1996.

[131] L. G. Valiant. A Bridging Model for Multi-core Computing. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA '08, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.

[132] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards A Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[133] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Commun. ACM*, 39(11):78–85, November 1996.

[134] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, pages 25–32, New York, NY, USA, 1996. ACM.

[135] T. Hoefler, L. Cerquetti, and F. Mietke. A Practical Approach to the Rating of Barrier Algorithms Using the LogP Model and Open MPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*,

ICPPW '05, pages 562–569, Washington, DC, USA, 2005. IEEE Computer Society.

[136] D.E. Culler, L.T. Liu, R.P. Martin, and C.O. Yoshikawa. Assessing fast network interfaces. *Micro, IEEE*, 16(1):35–43, 1996.

[137] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model — one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[138] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71 – 79, 1997.

[139] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 133–142, New York, NY, USA, 2001. ACM.

[140] J. L. Bosque and L. Pastor. A Parallel Computational Model for Heterogeneous Clusters. *IEEE Trans. Parallel Distrib. Syst.*, 17:1390–1400, December 2006.

[141] Z. Li, P. H. Mills, and J. H. Reif. Models and Resource Metrics for Parallel and Distributed Computation. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, HICSS '95, pages 51–60, Washington, DC, USA, 1995. IEEE Computer Society.

[142] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with Parallel Block Transfer. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 159–169, New York, NY, USA, 1990. ACM.

[143] C. A. Moritz and M. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):404–415, 2001.

[144] T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *In Proc. International Parallel and Distributed Processing Symposium (IPDPS 2000)*, pages 492–499, Cancun, Mexico, 2000. IEEE.

[145] F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg. HiHCoHP: Toward a Realistic Communication Model for Hierarchical HyperClusters of Heterogeneous Processors. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 42–48, Washington, DC, USA, 2001. IEEE Computer Society.

[146] A. L. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *Proceedings of the 3rd IEEE International Conference on Cluster Computing*, CLUSTER '01, pages 124–132, Washington, DC, USA, 2001. IEEE Computer Society.

[147] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. New Challenges in Dynamic Load Balancing. *Appl. Numer. Math.*, 52(2-3):133–152, February 2005.

[148] J. Faik, J. D. Teresco, K. D. Devine, J. E. Flaherty, and L. G. Gervasio. A Model for Resource-aware Load Balancing on Heterogeneous Clusters. Technical Report CS-05-01, Williams College Department of Computer Science, 2005.

[149] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

[150] A. Zavanella. *Skeletons and BSP: Performance Portability for Parallel Programming.* PhD thesis, UNIPI, December 1999.

[151] F. Gava. BSP Functional Programming: Examples of a Cost Based Methodology. In *Proceedings of the 8th International Conference on Computational Science, Part I*, ICCS '08, pages 375–385, Berlin, Heidelberg, 2008. Springer-Verlag.

[152] D. Pasetto and M. Vanneschi. Machine-independent Analytical Models for Cost Evaluation of Template-based Programs. In *Proc. of Intl. Euromicro PDP: Parallel Distributed and Network-based Processing*, pages 485–492, London, UK, January 1997. IEEE.

[153] R. Rangaswami. Compile-Time Cost Analysis for Parallel Programming. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 417–421, London, UK, 1996. Springer-Verlag.

[154] R. Rangaswami. *A Cost Analysis for a Higher-order Parallel Programming Model*. PhD Thesis. University of Edinburgh, Department of Computer Science, 1996.

[155] R. S. Bird. Algebraic Identities for Program Calculation. *Comput. J.*, 32(2):122–126, April 1989.

[156] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[157] T. A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, November 1994.

[158] D. G. Lowe. Object Recognition from Local Scale-Invariant Features. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2*, page 1150, Washington, DC, USA, 1999. IEEE Computer Society.

[159] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[160] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 176–185, Washington, DC, USA, 2011. IEEE Computer Society.

[161] X. Yang, X. Liao, K. Lu, Q. Hu, J. Song, and J. Su. The TianHe-1A Supercomputer: its Hardware and Software. *J. Comput. Sci. Technol.*, 26(3):344–351, 2011.

[162] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An Efficient, Model-based CPU-GPU Heterogeneous FFT Library. volume 0, pages 1–10, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[163] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 19–28, Washington, DC, USA, 2010. IEEE Computer Society.