Centro de Investigación y de Estudios
Avanzados del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

XSCALA: A Framework for Supporting Parallel Task Programming on Hybrid Heterogeneous Computing Systems

TESIS

Que presenta

M. en C. José Uriel Cabello Sánchez

Para obtener el grado de

Doctor en Ciencias
en Computación

Directores de la Tesis:

Dr. José Guadalupe Rodríguez García
Dr. Amilcar Meneses Viveros

Ciudad de México **Febrero, 2017**

Resumen

XSCALA: Un marco de desarrollo para soportar programación paralela por tareas en sistemas de cómputo híbridos hetrogéneos.

por Uriel Cabello

El avance en la comprensión y la solución de algunos problemas de investigación está estrechamente relacionado con nuestra capacidad de procesar grandes cantidades de datos lo más rápido posible. Por otro lado el uso de clústeres de computadoras de memoria híbrida equipados con aceleradores heterogéneos se ha convertido en la herramienta estándar más rentable para resolver dichos problemas dentro de periodos de tiempo razonables.

Hoy en día cada nodo en un clúster de computadoras está equipado no sólo con los tradicionales CPU, sino con otras unidades de porcesamiento de propósito general como los son las unidades de procesamiento gráfico (Graphic Processing Units ó GPUs) o los coprocesadores que tienen múltiples núcleos conectadas en configuraciones especiales, diseñadas para acelerar la ejecución de tareas simples pero repetitivas.

El enfoque actual para la programación de aplicaciones para sistemas híbridos heterogéneos requiere la combinación de múltiples modelos de programación y una cuidadosa distribución de la carga de trabajo.

Entre los principales problemas encontrados en las aplicaciones desarrolladas con el enfoque actual encontramos:

- Gestión compleja y subutilización de los recursos de cómputo.
- Aplicaciones fuertemente vinculadas a entornos de ejecución específicos.
- Poca abstracción del hardware subyacente.

En esta tesis presentamos XSCALA: Un marco de desarrollo para soportar programación paralela por tareas en sistemas de cómputo híbridos hetrogéneos. Los principales objetivos de XSCALA son dos: Simplificar el diseño y la implementación de aplicaciones aprovechando las ventajas de los sistemas híbridos heterogéneos y asegurar la escalabilidad de las aplicaciones que se están desarrollando.

Nuestro marco está diseñado para permitir la creación y distribución de tareas y está compuesto por una interfaz de programación de aplicaciones (Application Programming

Interface ó API) y un *middleware* que proporciona servicios en tiempo de ejecución como distribución de datos y calendarización.

Con el fin de demostrar las ventajas de nuestra propuesta presentamos la implementación basada en tareas de algunas aplicaciones. Realizamos varios experimentos que varían el tamaño del problema, la técnica de calendarización y el entorno de ejecución sin requerir modificaciones ni recompilación del código fuente programado con XSCALA.

Abstract

XSCALA: A Framework for Supporting Parallel Task Programming on Hybrid Heterogeneous Computing Systems

by Uriel Cabello

The advance in the understanding and solution of several research problems is closely related with our ability to processing large amounts of data as fast as possible. On the other hand the use of hybrid memory cluster computers with heterogeneous accelerators has become the standard and most cost-effective tool for solving those problems within reasonable amounts of time.

Nowadays each node in a cluster computer is equipped not only with the traditional CPU, but with other general purpose computing units like graphical processing units (GPUs) or coprocessors having multiple cores wired in special configurations, designed to accelerate the execution of simple but repetitive tasks.

The current approach for programming applications on hybrid heterogeneous systems requires the combination of multiple programming models and a careful distribution of the workload.

Among the major problems found in the applications developed with the current approach we find:

- Complex management and under-utilization of computing resources.
- Applications strongly tied to specific execution environments.
- Poor abstraction of the underlying hardware.

In this thesis we present XSCALA: a framework for supporting parallel task programming on hybrid heterogeneous computing systems. The major objectives of XSCALA are twofold: Simplify the design and the implementation of applications harnessing the advantages of hybrid heterogeneous systems and ensure the scalability of the applications being developed.

Our framework is designed to enable the creation and distribution of task and is composed of an application programming interface and a middleware that provides runtime services like data distribution and scheduling.

In order to demonstrate the advantages of our proposal we present the task-based implementation of some applications. We performed several experiments varying the size of

the problem, the scheduling technique, and the execution environment without requiring neither modifications nor recompilation of the applications code programmed with XSCALA.

Agradecimientos

A mis asesores por sus valiosas aportaciones.

Al CINVESTAV por las cátedras, la infraestructura y los apoyos otorgados que fueron invaluables en mi formacion.

Al CONACYT por la beca otorgada para la realización de esta tesis.

Contents

Chapter 1

Introduction

The use of hybrid memory cluster computing systems with heterogeneous accelerators has become the standard and most cost-effective approach for solving large scale scientific problems within reasonable amounts of time.

An hybrid memory cluster consists of several computers interconnected by a local network where each computer is equipped with one or more multicore processors. Each computer might be equipped with other types of general purpose processing units like GPUs or coprocessors accelerating the execution of certain tasks. The term hybrid refers to the combination of shared and distributed memory architectures commonly found in a cluster of workstations with multicore processors whereas the heterogeneity refers to the different capabilities and architecture of each processing unit. We will refer to this kind of system as an Hybrid Heterogeneous (H/H) computing system.

Exploiting H/H systems is a complex task that requires the combination of multiple programming tools and a careful distribution of the workload. For example: the shared memory model requires of tools like OpenMP or Pthreads, the distributed memory model requires of a communication middleware like MPI, and finally tools like CUDA or OpenCL are required to harness data parallelism in GPUs.

On the other hand, the performance achieved in the execution of an application is strongly related with the technique used for the distribution of the workload. Such distributions must avoid overloading a single computing device beyond its capabilities nor allocating tasks requiring intensive data communication in remote nodes connected through low bandwidth channels.

The problem of mapping tasks to devices in an optimal way is known as the scheduling problem and due to its complexity is strongly believed that there no exists an algorithm that can find optimal mappings using reasonable amounts of time. Several algorithms

based on heuristics have been developed for specific cases providing good solutions in short amounts of time. A heuristic for scheduling will try to find a workload distribution that minimizes the total execution time using some fixed rules.

This complex scenario raises the necessity for a new programming model that can provide a higher level of abstraction of the underlying hardware hiding the complexity of H/H architectures.

The new programming model must be supported by libraries, compilers and runtime execution environments working together to simplify the development of applications and to provide services for the execution of applications.

A set of tools, models and strategies designed to solve an specific problem is referred to as *"framework"* and can fill the gap between efficient use of resources and ease of programming.

Some of the frameworks found in the literature [Kegel et al., 2012],[Barak et al., 2010], [Alves et al., 2013], [Takizawa et al., 2013], [Aoki et al., 2010], [Augonnet et al., 2011] extend the capabilities of OpenCL, a low-level API for heterogeneous computing, to support transparent internode communications trying to emulate a single "super node" with many devices attached to it. This kind of work are built on top of distributed shared memory systems. Among the major disadvantages of this approach is that distributed shared memory systems are known to be slower than asynchronous message passing systems [Kshemkalyani and Singhal, 2011], besides of this, many useful services provided by message passing middleware are hidden for example, distributed file management, collective communication routines and distributed atomic operations, those services result essential to perform coarse data distribution in cluster systems. Finally it is worthwhile to say that the OpenCL API is too verbose and complex to use.

Another approach found in the literature consists in extend the capabilities of the middleware, in particular, of MPI to ease the management of GPUs and coprocessors [Lawlor, 2009], [Song and Dongarra, 2012], [Gabriel et al., 2004a]. These works can perform efficient data transfers using sophisticated communication hardware in a transparent way to the programmer. However a major disadvantage of these works is that the scheduling problem is still delegated to the programmer who is also responsible for coordinate the execution of the application.

Finally there exists some extensions to other conventional tools for multithreading programming like OmpSS [Fernández et al., 2014], and OpenACC [El-Ghazawi et al., 2005a] that rely on compiler directives, code annotations, APIs, and runtime systems to simplify the development of applications that can readily offload parts of the application to

accelerators however they are limited to shared memory (only one computer) limiting the scalability of the applications.

With respect to the scheduling problem some works [Augonnet et al., 2011], [Grasso et al., 2014], [Vasudevan et al., 2013] implement a dynamic scheduling technique to distribute the workload. In dynamic scheduling the tasks are dispatched at runtime using a centralized pool of tasks where the scheduler can distribute the work using fixed policies for example, finding the fastest computing device or the one that minimizes data transfer costs. Although this strategy has been thoroughly studied before [Cybenko, 1989],[Maheswaran et al., 1999], [Shirahata et al., 2010], [Cederman and Tsigas, 2011], and has proved to be very efficient in getting mappings it also have some disadvantages that must not be overlooked for example, a dynamic scheduler cannot anticipate future workloads which might in turn rise a performance degradation due to the possibility of premature overloading of the faster computing device, furthermore, the scalability of the applications is limited by the capabilities of the scheduling server.

The hypothesis that we hold in this work is that the parallel task programming model is a better approach to deal with the difficulties arising in H/H programming pointed out before. To assert the validity of this statement we present XSCALA (***X****platform* ***SCALA****ble*) a framework to support parallel task programming.

The objectives of XSCALA are twofold: simplify the development of applications adaptable to different computing environments and scalable as more hardware is used, and provide the mechanisms to ensure an efficient use of the computing resources.

To address the problem of load balancing we propose the integration of multiple scheduling strategies, in particular, we explore the advantages of static scheduling algorithms. Static scheduling can improve the performance of the applications with respect to dynamic scheduling due to the possibility of combine the analysis of the data dependencies with benchmark information to propose a plan of execution. Another advantage of static scheduling is a better use of the memory resources, a scarce resource that must be used and freed as soon as possible to use it in other tasks.

The major disadvantage of static scheduling is the large amount of information required to execute the algorithms. To overcome this problem all information required to perform static scheduling is automatically collected from the source code of the applications using a static analyzer plugin and from other performance evaluations for example: the compute capability of each device, latency in communications, and the bandwidth between each pair of processing units.

Finally to satisfy the trade-off between ease of programming and efficiency our framework introduces an API for task programming with functions resembling the MPI API which eases the learning process.

In order to demonstrate the advantages of the XSCALA framework and the degree of success achieved in our objectives we implemented all the components of the framework as well as several applications to measure the performance, overhead, and ease of programming achieved with our solution.

The major contributions of this work to the state of the art are the design of the framework architecture for supporting parallel task programming in hybrid heterogeneous environments, and the integration of new algorithms for the solution of concurrency and scheduling problems.

Among the applications used to test the framework we have linear algebra operations and the N-Body problem. We executed those applications using multiple platforms representative of the execution environments supported by XSCALA.

We compared the lines of code required for the implementation against the traditional approaches as well as the performance achieved in each execution. We found that using our tool the number of code lines is dramatically reduced while eases considerably the distribution of workload bringing portability to the applications. We also found that the scalability is achieved when the applications have the size or the granularity enough to hide latency cost.

This work is organized in three parts: context (chapters 2 and 3), contributions (chapters 4, 5, and 6), and results (chapter 7).

In chapter 2 we present the basic concepts and notation related to the use of parallel computers and the problems arising in this field. In the chapter 3 we review the state of the art in approaches for hybrid heterogeneous programming emphasizing on the relevant features of each work.

In the chapter 4 we present the architecture of the XSCALA framework including a description of each layer, its objectives, and its internal architecture. Next we presents our concept of the parallel task programming model and the functions defined in our API.

In chapter 5 we present the internal architecture of the middleware layer, its components and the algorithms implemented to solve the concurrency problems inherent in parallel programming. We also present formal definitions of the issues that entails to support the parallel programming model.

In chapter 6 we present our approach to solve the scheduling problem for H/H architectures with resource constraints. We present the scheduling algorithms proposed including a discussion of its complexity.

In the chapter 7 we present the implementation of some applications including linear algebra, the N-Body problem. There we show how to use our framework, and the reduction in the number of code lines. We performed several experiments varying the size of the problem and the scheduling technique.

Finally in the chapter 8 we present the conclusions and the future directions of this work.

Part I

Context

Chapter 2

Background

In this chapter we present a brief description of the concepts required to understand the scope and context of this work. Our intention consists in give a landscape view of what parallel programming for hybrid heterogeneous systems means and how can it be accomplished.

2.1 Basic Concepts

In this section we present the definitions of middleware and framework used to describe the components in our proposal and the concepts of scalability and portability used to understand the features desired in this work.

2.1.1 Middleware

Middleware is a software layer designed to provide an abstraction of another underlying software enabling the development of modular system architectures [Krakowiak, 2007]. There exists several types of middleware designed for specific purposes for example: message oriented middleware, grid middleware, database middleware, QoS middleware, remote procedure call (RPC) middleware, object request brokers (ORBs), etc.

Middleware is fundamental in the context of distributed applications where provides distributed services like naming, messaging, fault tolerance, process spawning, among others with the aim of enable multiple processes running on one or more machines to interact across a network. The middleware layer in distributed systems sits above the network operating system and below the application hiding the heterogeneity of the communication system [Bernstein, 1996].

RPC systems and ORBs middlewares deal with the distribution of the workload by requesting the execution of procedures or methods to other peers in the network. This approach results specially beneficial for applications that can delegate the execution of the hardest parts to other specialized peers.

Message oriented middleware employs peer-to-peer relationships between individual peers, where each peer can send and receive messages to and from other peer [Curry, 2004]. Message oriented middleware can be implemented using a message delivering approach like MPI. This approach is suitable to solve scientific problems dealing with big amounts of data, letting each node work with a chunk of the data.

2.1.2 Frameworks

A Framework is an integrated set of components that collaborate to provide a reusable solution for a family of related problems. A formal definition of the concept is given by Smith [Schmidt et al., 2000] that defines a framework as a concrete realization of patterns that facilitate direct reuse of a detailed design. In this definition a pattern describes a particular recurring problem that arises in specific contexts and a well proven generic scheme for its solution *i.e.* the "best practices" for solving the problem. The solution scheme is specified by describing its constituent components, their responsibilities and relationships.

Among the components integrating a framework we have:

- Application programming interfaces.
- Libraries.
- Runtime systems.
- Middlewares.
- Compilers, wrappers and pre-processors.

An Application Programming Interface (API) is a flat collection of function specifications, wrapping routines, and protocols with frequently used functionality [Buschmann et al., 1996]. APIs usually can be found in the form of header files and documentation.

Libraries and runtime systems are implementations of the functions described in the API but libraries differ from runtime systems in that the former performs a specific function when is requested by the main program while the latter provides a set of functions executed before, during, and after the execution of the main program. The functions

contained in the libraries and runtime systems can be requested by the applications using the API, although some of the functions might be hidden to the programmer and are used internally to provide services like task scheduling, resource management, among others.

Some frameworks include tools to build the applications for example, compilers, wrappers and other code preprocessors. Some examples of this kind of tools are the *Nvidia c compiler* [Nickolls et al., 2008], the *OpenMPI c compiler* [Gabriel et al., 2004a].

2.1.3 Portability

Portability refers to the degree to which an application can be moved from one execution environment to another achieving the same results [Marowka, 2010]. In the high performance computing field the portability is also related with the capability to achieve similar performance when the application is executed in different environments with similar capabilities, this kind of portability is refereed to as performance portability.

2.1.4 Scalability

Scalability is the capability to get an increase in performance of a specific application as more hardware resources are added [El-Rewini and Abd-El-Barr, 2005]. There exists two types of scalability:

- Strong scalability: The total problem size stays fixed as more processors are added.
- Weak scalability: The problem size per processor stays fixed as more processors are added.

Scalability is one of the most important nonfunctional requirements for several applications of HPC given that enables harnessing all the resources available for the execution.

2.2 Parallel Computer Memory Architectures

In this section we present a classification of parallel computers based on their memory architecture. We present the most remarkable features of each class of memory architecture namely: shared memory, distributed memory, and hybrid memory.

2.2.1 Shared Memory Architectures

Computers with shared memory architecture have a common memory accessible to a number of physical processors interconnected by fast local buses (e.g. cross bar or QPI). Communication between tasks running on different processors is performed by writing and reading to the global memory space [El-Rewini and Abd-El-Barr, 2005].

There exists two classes of shared memory architectures: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) [Herlihy and Shavit, 2012]. Computers having one multicore processor are examples of UMA architectures where multiple identical "*cores*" have equal access time to the shared memory space. A computer having two or more processors on the same board is an example of NUMA architecture where each core of each processor have fast access to "*near*" shared memory space and slow access to "*remote*" shared memory space.

2.2.2 Distributed Memory Architectures

Distributed memory architectures are systems composed of a number of networked computers each with its own local processing resources and separate memory spaces using a communication protocol to share data and coordinate their actions [Coulouris et al., 2011]. An example of this type of architecture are commodity clusters, which consists of several workstations connected by a local area network (LAN) using the message passing interface (MPI) to share data. This approach has proven to be a cost-effective solution for certain problems of large scale like simulations or climate forecasting.

2.2.3 Hybrid Memory Architectures

Hybrid memory architecture refers to the combination of shared and distributed memory architectures [Herlihy and Shavit, 2012]. This type of architectures are built by connecting several workstations in a LAN with each computer having shared memory architecture itself for example, it might have one or more multicore processors.

2.3 Heterogeneous Computing Architectures: GPUs and Coprocessors

A cluster computer is heterogeneous when its processing units have different characteristics or capabilities. This diversity is the result of the accumulation of multiple computing resources some of them specialized in the execution of some tasks.

The objective behind this accumulation of resources is to reduce the time required to get a solution (strong scalability) or to solve bigger problems within a reasonable amount of time (weak scalability) [Dongarra and Lastovetsky, 2009]. A cluster of computers is defined as heterogeneous if some of the following conditions are meet:

- Processors in the cluster are not identical neither in architecture nor in capabilities.
- The communication network may have a regular architecture but with heterogeneous components. For example, it might consists of a number of faster communication segments interconnected by relatively slow links. Such a structure can be obtained by connecting several homogeneous clusters in a single multicluster.
- The cluster may be a multitasking computer system, allowing several independent users to run simultaneously their applications on the same set of processors (but still dedicated to high-performance parallel computing).

Nowadays, each node in a cluster-computer is equipped not only with traditional CPUs but with other general purpose computing units like graphical processing units (GPUs) or coprocessors having multiple cores wired in special configurations, designed to accelerate the execution of certain tasks.

GPUs are computing devices capable of running thousands of lightweight threads in parallel [Kirk and Hwu, 2016]. The typical architecture of a GPU is depicted in Figure 2.1. GPUs are specially suited for the execution of simple but repetitive tasks due to the presence of hundreds or thousands of cores managed from a single control unit. In order to exploit the computing power of GPUs the programmer is encouraged to make a careful planning of the work avoiding thread divergence and coordinating the access to multiple memory hierarchies with different sizes and speeds.

Coprocessors are other type of general purpose computing device similar to GPUs but with a smaller amount of cores, instead, coprocessors have tens of independent cores capable of running more complex and even divergent tasks [Jeffers and Reinders, 2013]. Each core have fast access to large amounts of local memory within the device and larger amounts of cache than in the GPUs. The architecture of a coprocessor is depicted in Figure 2.2.

GPUs and coprocessors have its own Instruction Set Architecture (ISA) requiring the use of cross compilers to generate machine code suitable for each architecture. The necessity of building machine code specific for each architecture rises a serious limitation to the portability of the applications thus requiring recompilation of applications and the use of specific development tools for each vendor.

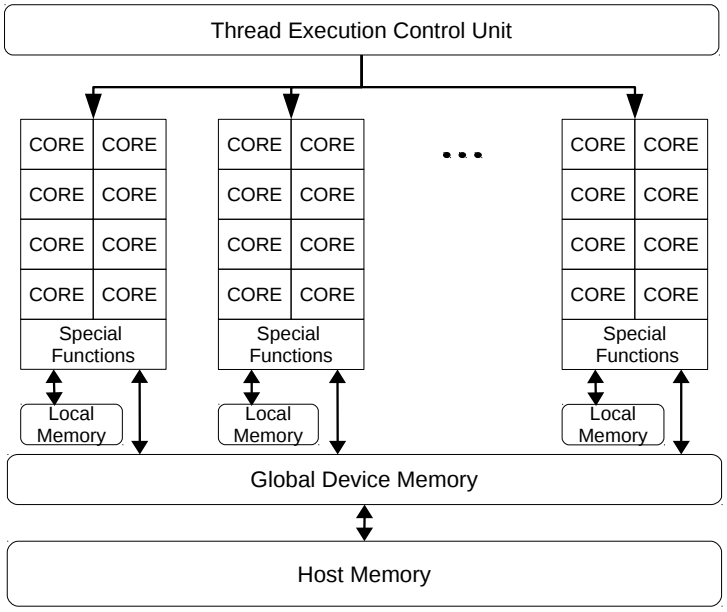


Figure 2.1: Architecture of a GPU.

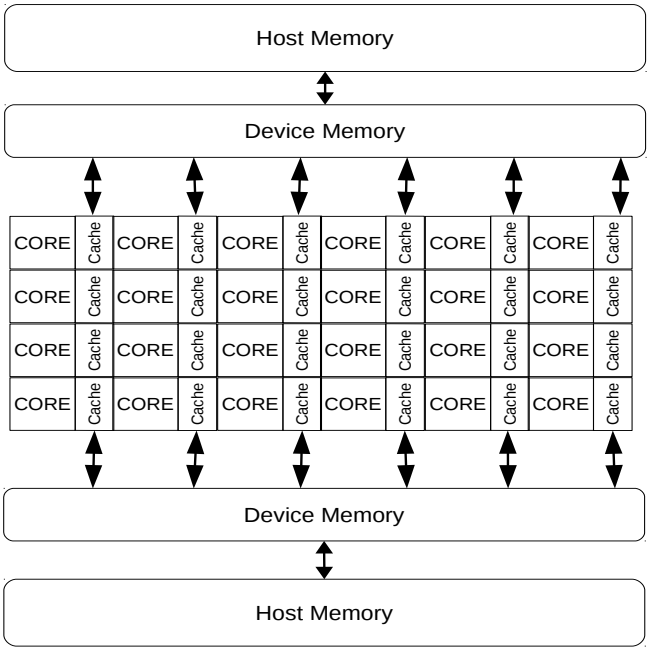


Figure 2.2: Architecture of a coprocessor.

Hereinafter we will use the terms *"host"* or *"node"* interchangeably to refer to each one of the computers in the cluster and the terms *"processing unit (PU)"*, *"processing element (PE)"* or *"device"* will be used indistinctly to refer to any processing unit regardless of its architecture. The concept of heterogeneity will refer to both: the diversity in the capabilities of each device and the diversity in the instruction set of the processing units. Finally the term hybrid heterogeneous (H/H) cluster will refer to a cluster of computers interconnected through a local network with each node equipped with one or more heterogeneous processing units.

An example of H/H system is depicted in Figure: 2.3 with two nodes connected through a network device where each node have two processing units: a multicore CPU, and a many core accelerator. The major advantage of this kind of architectures is that they can be readily extended to thousands of nodes in the network with several CPUs or accelerators for each node.

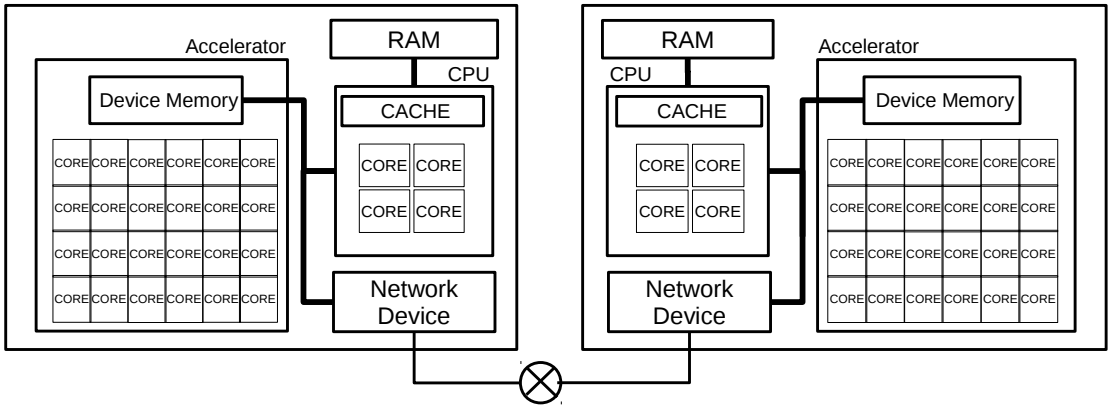


Figure 2.3: Architecture of an hybrid heterogeneous (H/H) computing system.

2.4 Parallel Programming Models

In its more general form a *programming model* consists in the definition of certain structures and their relationships to provide an abstraction of the underlying hardware with the aim of ease the design and implementation of algorithms.

The major objective of a programming model is to let the programmer or the algorithm designer to assume that certain events just happens and do not be carried about how they actually happen. For example, in task programming the programmer can assume that each task will be eventually executed in a processing unit no matter which and that the data transfers will be eventually completed no matter how.

Parallel programming models are a class of programming model specially useful in the design of applications where it is possible to take advantage of the replication and distribution of work among several execution units with the objective of achieve a faster execution. Each model might in turn define its own execution units for example threads or processes.

Parallel programming models can be classified based on the abstraction of memory hardware that they make [Balaji, 2015] i.e shared memory, distributed memory or hybrid memory. This classification and some examples of each class is depicted in Figure: 2.4.

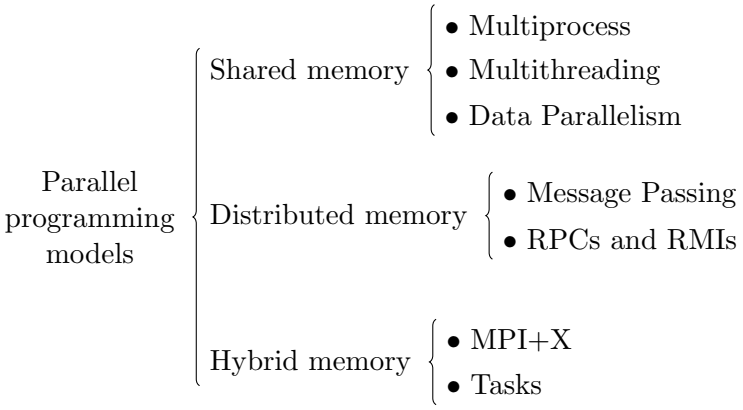


Figure 2.4: Classification and examples of parallel programming models.

2.4.1 Shared Memory Programming Models

This class of programming models are specially designed for shared memory architectures where each execution unit have a local unified view of all the data of the application.

Multiprocess and Multithreading

Multiprocess and multithread programming are two types of programming models based on the idea of let two or more execution units namely: processes in the case of the former and threads in the case of the latter to interact along the execution of the application [Raynal, 2012], [Herlihy and Shavit, 2012]. Therefore, the processes of a multiprocess application and the threads in a multithread program may execute simultaneously ("in parallel" or "concurrently") parts of the same program. This two programming models are strongly related due to the problems and techniques used to solve synchronization issues hence they are commonly refereed to as "concurrent programming"

The major concern in concurrent programming is the problem of synchronization that occurs when the progress of one or several processes (threads) depends on the behavior

of other processes (threads). More generally, synchronization is the set of rules and mechanisms that allows the specification and implementation of sequencing properties on statements issued by the processes (threads) so that all the executions of a multiprocess (multithread) program are correct [Raynal, 2012].

Concurrent programming is specially suited for shared memory architectures where the processes (threads) have fast and secure access to several forms of locking and synchronization structures (e.g., mutexes, semaphores, or monitors) to coordinate the execution of the program. Some canonical programming tools supporting multithreading are OpenMP and Pthreads whereas multiprocess must be implemented using libraries and tools provided directly by the operating system.

Data Parallelism

The data parallel model also referred to as the Partitioned Global Address Space (PGAS) model consists in a set of execution units having access to a global address space, where each execution unit performs the same operation on a different partition of the data [Kirk and Hwu, 2016].

This model is well suited for problems exhibiting coarse grained parallelism where a large number of operations can be performed without requiring neither synchronization nor messaging between the execution units but must avoid task divergence.

This model requires the design of the applications in such a way that the selection of a block size do not compromises the correctness of the algorithm offering great scalability at the cost of making the programmer responsible for ensure the correctness of his program regardless of the distribution of the array between the execution units.

Some programming tools supporting data parallelism are *Unified Parallel C* (UPC) [El-Ghazawi et al., 2005b] and *thread building blocks* [Reinders, 2007].

2.4.2 Distributed Memory Programming Models

Message Passing

The message passing programming model consists in several networked processes using their own local memory space and communicating with each other using messages. Each process is attached to one computer and can be any number of physical computers hosting the processes (even only one) [Coulouris et al., 2011]. The message passing

model is specially suitable to achieve weak scalability due to the possibility to add more computers to the network in order to solve problems with more data.

Communication in this model is typically done using send/receive pairs of commands, broadcasting and other communication primitives but there is no concept of global memory space, therefore, synchronization is also performed using messages from one local memory space to all others making this process slow and more complex than in the shared memory approach.

RPC and RMI

Remote procedure call (RPC) and remote method invocation (RMI) are other two programming models used to achieve parallel execution of work. An application using remote objects is typically organized in a client-server approach: a process or thread executing a method of a client object sends a request to the server object to execute a method of that object [Tanenbaum and van Steen, 2007], [Coulouris et al., 2011].

Although RPC and RMI were not specifically designed for parallel programming they can achieve parallelism by letting the caller to perform multiple calls to the server (or create several objects in the case of RMI) while the server side administrates several parallel execution units serving the requested methods. This programming model is used in works like charm++ Kale and Zheng [2009] enabling it to provide other useful services like fault tolerance and load balancing.

2.4.3 Hybrid Memory Programming Model

A hybrid programming model have the objective of exploit the advantages of multiple models by combining some of them. In particular, the combination of the message passing model with other model for shared memory programming is referred to as the MPI+X programming model [Quinn, 2004].

The hybrid programming model is currently the most common approach to harness H/H systems. This model consists in a coarse separation of the data to be processed among multiple MPI processes located in different machines followed by groups of threads in GPUs or coprocessors performing computationally intensive tasks using local on-node data. Some typical examples of the tools used to support hybrid programming models includes MPI+OpenMP, MPI+CUDA, MPI+OpenCL, and MPI+OpenMP+CUDA.

The hybrid programming model is depicted in the Figure: 2.5 where we show two nodes executing one application that offloads parts of the computation to the devices. The

transfers of data from the host to the device (H2D) and conversly (D2H) is requested by the application as well as the send and receive operations for internode communications.

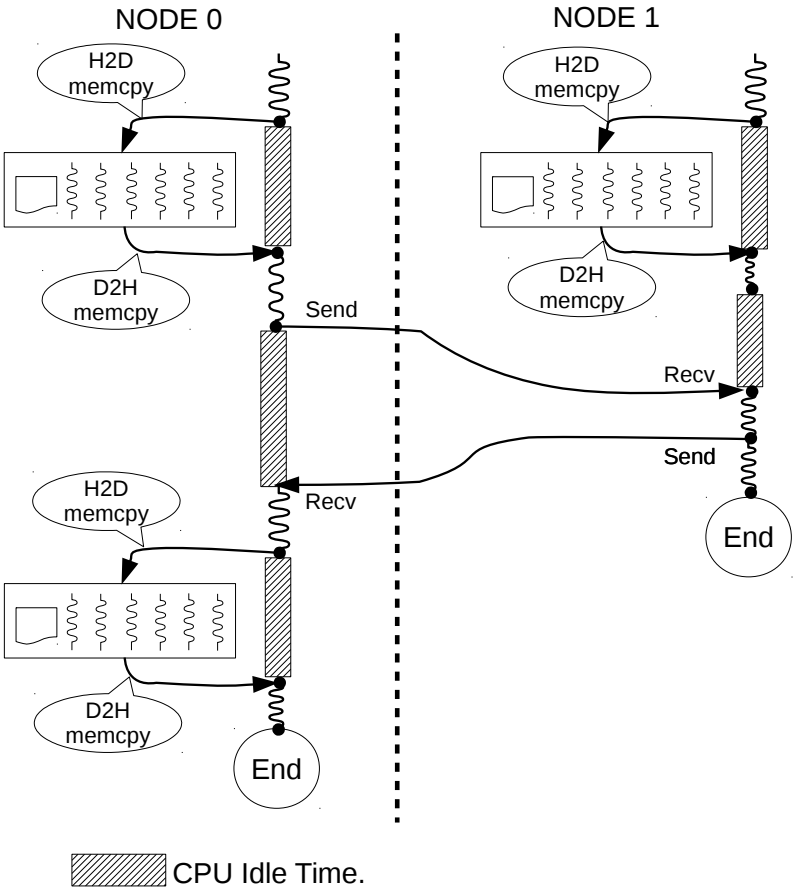


Figure 2.5: MPI+X procedural programming model. Portions of the computation are offloaded to the processing units using explicit data transfers and execution requests. Internode communications are completed through messages.

The major disadvantage of the hybrid approach is the complexity associated with the combination of multiple tools as well as the difficulty to achieve an efficient execution. The use of synchronous calls in the delegation limits the scalabilty of this approach to one device per node. When multiple devices are attached to the node the model requires an additional tool to implement threads and callback functions to enable the use of asynchronous calls.

2.5 The Parallel Task Programming Model

The task programming model is a high level programming model based on the construction and distribution of small units of work called *"tasks"*.

The most remarkable feature of task programming is that tasks are self-contained, in the sense that they can be executed and completed by a batch of threads entirely without

requiring intervention of other tasks. The tasks are linked through data dependencies establishing in this way the order of execution.

The task programming model can take full advantage of H/H systems by exploiting both: coarse grained parallelism and fine grained parallelism. The former is achieved by launching multiple tasks to be executed concurrently or in parallel using multiple devices and the latter consists in launching several threads using the Single Instruction Multiple Data (SIMD) model on each task to complete the work. This model is refereed to as parallel task programming and is depicted in the Figure: 2.6.

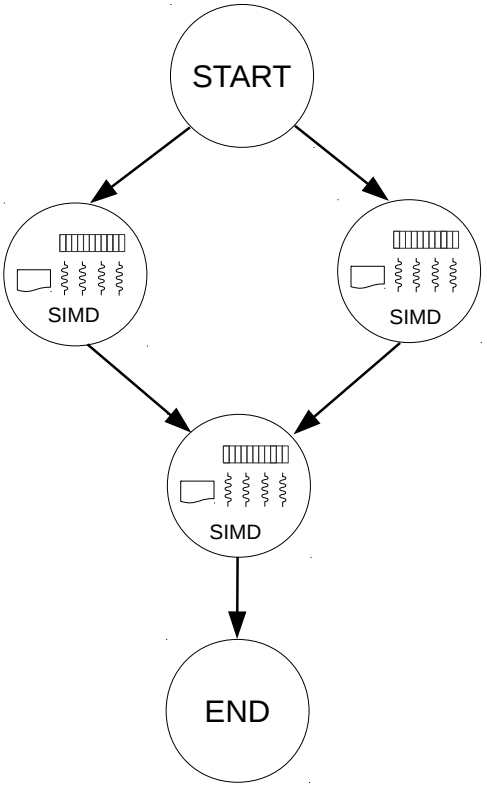


Figure 2.6: The task programming model. Data is transferred automatically between the devices regardless of their location and a scheduler can assign the tasks to the most suitable device.

Unlike the MPI+X approach where the programmer is responsible for offloading portions of the application into specific processing units and take care of synchronization issues the parallel task approach provides a higher abstraction of the underlying computing system handling many of these problems in a transparent and efficient way for example, the data transfers are overlapped with the computation and the tasks are assigned to the most suitable device.

2.5.1 History and Advances

The adoption and implementation of the task model is relative more recent than other paradigms like structured and object oriented programming. This is partly explained because they were expressive enough for the bast majority of algorithms while provided an enough level of abstraction of the most common computing architectures.

In 1993 Rinard presented JADE [Rinard et al., 1993], one of the first works in the task programming direction, remarking that some task may require special hardware to enable its execution or to accelerate its completion.

More recently the OpenMP standard advanced in the task programming trend. OpenMP is a specification[1] for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism. Since release 3.0 the OpenMP specification was extended to incorporate support for task-based parallelism in shared memory multiprocessor architectures [Ayguadé et al., 2008], [Ayguadé et al., 2009].

Another languages and execution environments have been extended to support task parallelism for example, the task parallel library in microsoft .NET [Leijen et al., 2009] or intel Thread Building Blocks (TBB) [Reinders, 2007], that are libraries supporting the task programming model in multicore processors.

2.5.2 Representation of Workflows with Graphs

The sequence in which the execution control is transferred from task to task is referred to as *workflow* and can be represented with graphs.

A graph G is composed of a non-empty finite set V(G) of elements called vertices and a finite set E(G) of distinct unordered pairs of elements of V(G) called edges [Wilson, 1996]. Therefore a graph G is defined as a pair (V, E) where V is called the vertex set and E the edge set. An edge is an unordered pair $\{v, w\}$ joining the vertices v and w of V, and we say that vertex v and w are adjacent if $\{v, w\} \in E$.

Directed graphs (or digraphs) differ from graphs in that the former is defined in terms of ordered pairs thus making the edge (v, w) different from the edge (w, v) when both are in E. In digraphs the out-degree of a vertex is the number of edges leaving it, and the in-degree of a vertex is the number of edges entering it [Cormen et al., 2001].

[1] OpenMP Architecture Review Board (Ed.) (2008, May). OpenMP Application Program Interface. Retrieved September 18, 2016, from http://www.openmp.org/wp-content/uploads/spec30.pdf Version 3.0

A weighted graph $G_c = \{V, E, c\}$ is a graph where each edge $e \in E$ has an associated weight given by a weight function $c : E \to \mathbb{R}$. The vertices in a graph can have weights instead of or besides to the weights of the edges [Wilson, 1996]. A graph having weighted vertices will be denoted as $G_w = \{V, E, c, w\}$.

We can represent the resources available in a computing system with the graph $G_r = \{V_r, E_r, c_r\}$ were each vertex $v \in V_r = \{PU_1, PU_2, ..., PU_n\}$ denotes a processing unit, each edge $e \in E_r = \{C1, C2, ..., C_k\}$ denotes a communication link between two processing units, and $c_r : E_r \to \mathbb{R}$ represents the bandwidth between each pair of PUs.

Similarly, we can use a weighted graph $G_t = \{V_t, E_t, c_t, w\}$ to represent the workflow of any given application. Here the vertex set V_t represents the set of task to be executed, the edge set E_t represents the data dependencies between tasks, $c_t : E_t \to \mathbb{R}$ represents the amount of data to be transferred between to tasks and $w : V_t \times V_p \to \mathbb{R}$ represents the cost of executing any give task of V_t in a processing unit of V_p.

Many scientific problems can be solved based on the definition of tasks and the accumulation of the results, furthermore there exists some basic patterns to express workflows [van der Aalst et al., 2003], and many others can be implemented using combinations of them.

Sequence

Sequence is the most common pattern and establish the consecutive steps in a workflow as is depicted in the figure 2.7. In this pattern the task j can start only after the arrival of the data coming from the task j. In this case we say that the task i is the predecessor of j or that the task j is the successor of i.

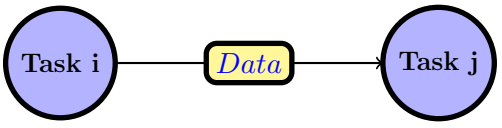


Figure 2.7: The sequential pattern.

Parallel Split

Parallel splits are points in the workflow where one executing task can activate the execution of more than one successors. The new activated tasks can be executed concurrently or in parallel as is depicted in the Figure: 2.8.

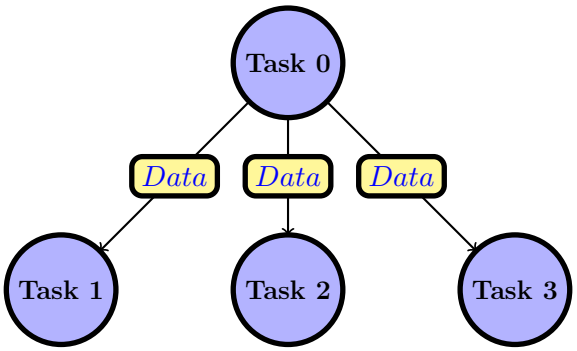


Figure 2.8: Workflow with a parallel split.

Synchronization

Synchronization are points in the workflow where multiple independent tasks converge without require additional data transfers among them. This pattern is depicted in figure 2.9. This pattern is useful to ensure that each task has reached certain state before to proceed to the next operation.

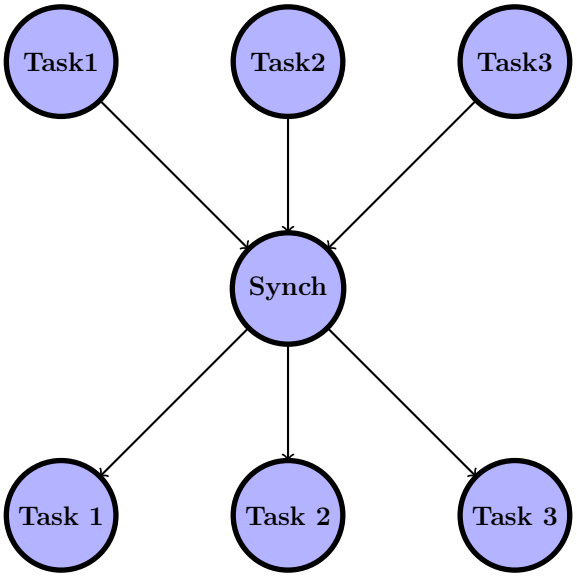


Figure 2.9: Workflow with a synchronization point.

Single Merge

Single merge patterns are points in the workflow where a single task have data dependencies form two or more branches of the graph. This pattern is depicted in figure 2.10.

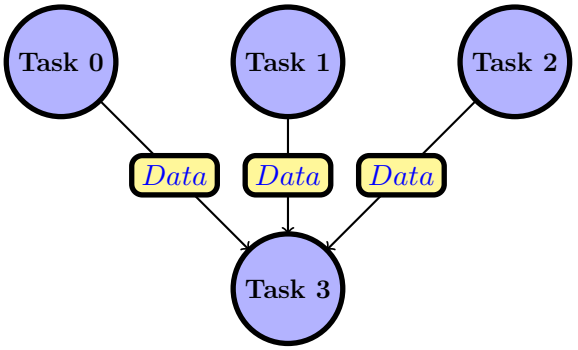


Figure 2.10: Workflow with a merge.

Multiple Merge

Multiple merge patterns are points closely related to single merges but in this case there exists two or more tasks in the DAG having common dependencies fulfilled at the same time as is depicted in figure 2.11.

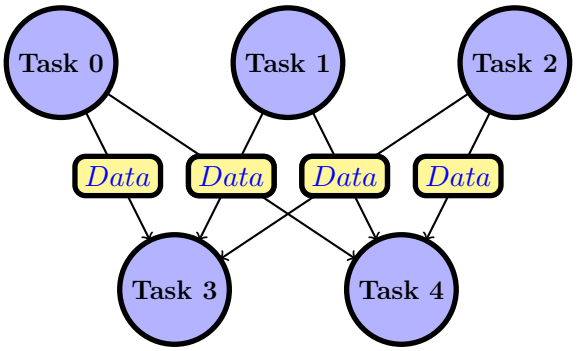


Figure 2.11: Workflow with multi merge.

2.6 Scheduling

The performance achieved in the execution of an application is strongly related with the technique used for scheduling tasks. In this section we present a formal definition of the scheduling problem and a classification of the scheduling techniques based on the features of the problem.

2.6.1 Classification of Scheduling Algorithms

Let G_r and G_t to represent the resource and task graphs as previously defined in section 2.5.2. A task mapping M is defined as a set of ordered pairs $\{(i, j) \mid i \in V_t, j \in V_p\}$

where each ordered pair (i, j) reflects the assignment of the task i to be executed in the processing unit j.

The task scheduling problem consists in finding the mapping M that minimizes the completion time of all the tasks in G_t subject to certain constraints. A naive approach to solve this problem is testing all combinations but it could take up to m^n evaluations with $m = |V_r|$ and $n = |V_t|$. This number of evaluations talks about the difficulty of the problem, furthermore, the general formulation of the task scheduling problem is known to be an NP-complet problem [Gary and Johnson, 1979].

The scheduling problem have several specific variants depending on the constraints, the number of processors and the features of the tasks, however, very few of them can find optimal solutions using polynomial time algorithms [Brucker, 2013]. Given the complexity of the problem many algorithms based on heuristics have been formulated to construct *"good"* mappings for complex cases of the problem using a reasonable amount of time.

Scheduling algorithms can be broadly divided depending whether the mapping decisions are taken at runtime (dynamic scheduling) or at compile time (static scheduling). The former is appropriate when neither the amount of task to be executed nor the data dependencies among them are known beforehand, and the latter can improve the application performance due to its ability to take mapping decisions based on the characteristics of the tasks, the data dependencies and the capabilities of the computing resources.

The classification of the scheduling algorithms can be further divided depending on whether they consider or not data dependencies, if preemption is allowed, or if they consider or not heterogeneous resources. Given the overwhelming amount of variations of the scheduling problem Graham [Graham et al., 1979] introduced a scheme known as three field notation $\alpha|\beta|\gamma$ to classify the scheduling problems.

This classification assumes that n Jobs $J_1 \ldots, J_n$ have to be processed on m machines $M_1, \ldots, M_m$, where each machine can handle at most one job at a time and each job can be executed by at most one machine at a time. Some possible values of the parameters α, β, γ are as follows:

The field α specifies the architecture of the computing system and have two components denoted by α_1, α_2. Some possibles values for α_1 are $\{P, Q, R\}$ where:

- $\alpha_1 = P$ indicates parallel identical machines.
- $\alpha_1 = Q$ indicates parallel uniform machines.
- $\alpha_1 = R$ indicates parallel unrelated machines.

And α_2 is an optional positive integer equal to m.

The field β is a list of values indicating job characteristics where: *"pmtn"* indicates if preemption is allowed, *"prec"* indicates precedence constrains, and *"1"* indicates that each operation has unit processing time.

Finally the third field denotes the optimality criterion. Some common criteria are:

- $C_{max} = max\{C1, \ldots, C_n\}$ (maximum completion time)
- $\sum C_j = C1 + \ldots + C_n\}$ (total completion time)

C_{max} is the most widely used criteria and is known as the *makespan* of the application.

Some representative scheduling algorithms based on heuristics includes the minimum completion time (MCT), the opportunistic load balancing (OLB) [Armstrong et al., 1998], minimum execution time (MET), the min-min heuristic [Braun et al., 2001], insertion scheduling heuristic (ISH), bubble scheduling and allocation (BSA) [Kwok and Ahmad, 1999], dynamic level scheduling (DLS) [Sih and Lee, 1993], modified critical path (MCP) [Wu and Gajski, 1990], heterogeneous earliest finish time (HEFT) [Topcuouglu et al., 2002], dynamic critical path (DCP) [Kwok and Ahmad, 1996], Highest Levels First with Estimated Times (HLFET) [Adam et al., 1974].

The classification of those algorithms and the problems that they solve is summarized in the Table 2.1.

	Scheduling problem	Scheduling algorithms
Dynamic Scheduling	$P\|\|C_{max}$	Round Robin
	$R\|\|C_{max}$	Work Stealing
	$R\|\|C_{max}$	Diffusion
	$Q\|\|C_{max}$	Priority Queue
Static Scheduling	$P\|prec, 1\|C_{max}$	Hu's algorithm
	$Q\|\|C_{max}$	OLB, MET, MCT, min-min
	$R\|prec\|Cmax$	HEFT
	$P\|prec\|Cmax$	ISH
	$P\|prec\|Cmax$	DCP
	$P\|prec, pmtn\|Cmax$	HLFET
	$R\|prec\|Cmax$	DLS

Table 2.1: Classification of Scheduling Algorithms

Static scheduling for tasks with data dependencies and without loops is a special case of the scheduling problem that has been thoroughly studied in the literature using priority lists. In spite of the complexity of the scheduling problem some algorithms based on priority lists [Hu, 1961], [Sih and Lee, 1993], [Yang and Gerasoulis, 1994], [Topcuouglu

et al., 2002] can obtain quasi-optimal schedules modeling the problem with directed acyclic graphs (DAGs) and sorting the tasks based on certain priority rules.

2.6.2 Scheduling Subject to Constraints

When the execution of a task is restricted by the availability of one ore more scarce resources we say that the scheduling problem is subject to resource constraints. Blazewicz [Blazewicz et al., 1983] proposed an extension to the three field classification schema to aggregate the restrictions in the availability of a resurce to the β parameter as "**res**$\lambda\sigma\varrho$" where:

- λ is a positive integer representing the number of scarce resources.
- σ is a constant positive integer indicating that all the resources have an equal initial size; when it is not explicitly indicated a matrix s must be provided in the problem specification to indicate the initial amount of each resource on each device.
- ϱ is a constant positive integer indicating that all resource requirements are equal in size; when it is not explicitly indicated an additional matrix r must be provided to indicate the requirement of each resource on each task.

2.7 Performance Indicators

Performance refers to the degree of success in the execution of a task with respect to certain measure [Dongarra and Lastovetsky, 2009], [McCool et al., 2012] for example: the rate at which tasks are computed (throughput) or the time that takes to complete a task (latency).

The principal objective to use parallel computers is to improve the performance of applications regardless of the specific measure. This implies an iterative process that requires the use of certain metrics to compare the degree of improvement achieved on each step.

The two metrics commonly used are speedup and efficiency. The speedup compares the time that it takes to solve certain problem using one execution unit against the time required to solve the same problem using P processing units, *i.e.*:

$$speedup = S_p = \frac{T_1}{T_p} \tag{2.1}$$

Where T_1 is the time required using one execution unit (also known as the sequential execution) and T_p is the time taken using P execution units.

The efficiency refers to the degree at which we are exploiting new added resources. Efficiency is computed dividing the speedup by the number of execution units i.e:

$$efficiency = \frac{S_p}{P} = \frac{T_1}{PT_p} \tag{2.2}$$

The use of processing units at full of its capacity yields an efficiency equal to one however this is seldom possible due to the overhead generated by communications and synchronization between the execution units as well as the existence of parts inherently sequential in the applications. The Amdahl law [Amdahl, 1967] establishes that exists an upper bound in the maximal speedup that can be achieved when adding resources to solve a problem given by:

$$S_p \leq \frac{1}{f + (1-f)/P} \tag{2.3}$$

Where f represents the fraction of sequential execution of the application and P the number of execution units. In particular when f tends to infinity we have that:

$$S_\infty = \frac{1}{f} \tag{2.4}$$

Which demonstrates the existence of such limit. Achieving the best performance subject to the Amdahl constraint represents a form of strong scalability.

On the other hand the Gustafson-Barsis' law [Gustafson, 1988] makes an important observation: "*the speedup must be measured by scaling the problem to the number of processors, not by fixing the problem size.*" This observation is interpreted as follows: let $W' = fW + (1-f)nW$ to be the workload of the problem with the parallelized fraction scaled by a factor of n thus the processing time using n execution units is given by:

$$T_n = \frac{fW}{1} + \frac{(1-f)nW}{n} = fW + (1-f)W \tag{2.5}$$

Therefore the theoretical speedup achieved using n processing units is given by:

$$S_n = \frac{fW + (1-f)nW}{T_n} = f + (1-f)n \tag{2.6}$$

Escaping thus the theoretical limit to the speedup imposed by Amdahl's law. The Gustafson-Barsis' law represents the weak scalability.

Chapter 3

The State of the Art

The development of applications able to harness H/H computing systems is a complex task requiring a deep knowledge of the architecture of the system being used and of multiple programming tools. This problem rises the necessity of new tools that can hide the complexity of the underlying system easing the development of portable applications that can scale as more hardware is added even if it is heterogeneous.

Among the proposals found in the literature to solve the H/H programming problem the most common consists on extend the capabilities of OpenCL to perform transparent communication between the processing units even if they are located on multiple hosts [Grasso et al., 2014], [Kegel et al., 2012], [Aoki et al., 2010], [Kim et al., 2012], [Alves et al., 2013], [Takizawa et al., 2013].

Other common approach is extending the capabilities of MPI to ease the management of heterogeneous computing devices [Lawlor, 2009], [Aji et al., 2012], [Song and Dongarra, 2012].

Finally some works proposed new APIs and compiler tools to address the programming problem pointed out above [Augonnet et al., 2011], [Vasudevan et al., 2013], [Elangovan et al., 2014], [Scogland et al., 2014]. The related works can be broadly classified based on the programming tools implemented in the proposal as is depicted in the Fig. 3.1. The details of each proposal are reviewed in the rest of this chapter.

3.1 Extensions to OpenCL

OpenCL (Open Computing Language) is a multivendor open standard for general-purpose parallel programming of heterogeneous systems that includes CPUs, GPUs and

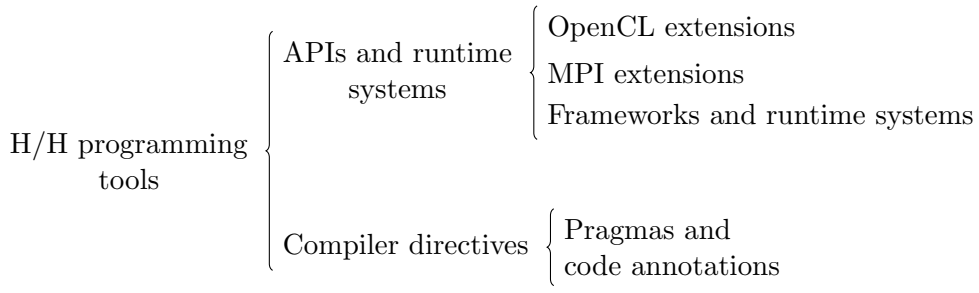


Figure 3.1: A classification of the proposals in the state of the art based on the programming tools implemented by each work.

other processors first released in 2009 [2].

The OpenCL API provides a standard mechanism for offloading portions of the computation over multiple heterogeneous computing devices. This is achieved using a set of handlers mapping the requests of the application to the OpenCL implementations regardless of the architecture of the device.

A typical application in OpenCL is composed by two kinds of code: host code and device code. The former includes all the API function calls to create the handlers, and the latter is code designed to harness data parallelism in the device.

The host code must follow the OpenCL specification that is defined in four parts:

Platform Model: Defines the abstract hardware model called "*Compute Device*", composed of "*Compute Units with*" (stream multiprocessors)having "*Processing Elements (cores)*"

Execution Model: Defines how the OpenCL environment is configured. Includes concepts like work group, work item, context objects etc.

Memory Model: Defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies and is depicted in the Figure: 3.2.

Programming model: Defines how the concurrency is mapped to physical hardware. OpenCL supports data parallel and task parallel programming models.

In spite of its advantages OpenCL has two strong disadvantages: The implementations are not interoperable, and it is not designed to scale beyond a single node. This occurs

[2]The Khronos Group (Ed.) (2009, October). The OpenCL specification. Retrieved September 18, 2016, from https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf Version 1.0

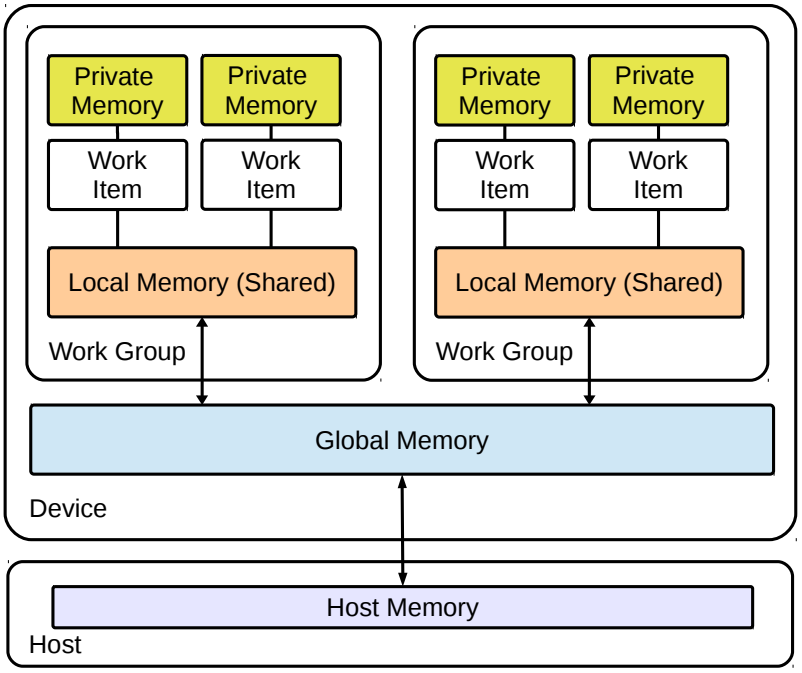


Figure 3.2: OpenCL memory model.

due to the fact that neither intercontext nor internode communications are part of the specification.

Some works found in the literature try to overcome those limitations with the aggregation of new functions to the standard API, as well as the implementation of runtime systems that can provide communication services required for internode data transfers or to request kernel executions on remote nodes.

The libWater library [Grasso et al., 2014] implements an event driven task programming model where the runtime system tracks dependency information to dynamically built a list of commands distributed to multiple local command queues. This approach enables the detection of the communication pattern to reduce the number of device-host-device data transfers. This library implements a device query language (DQL) used to ease the mapping of tasks to the computing device that meets some requirements. Although this approach eases considerably the management of computing devices the criteria used for the selection of the device is limited to the information provided by the driver of each device leaving aside another important features like, the compute capability, the internal and external bandwidth, or the latency between computing devices that undoubtedly have a major impact in the performance of the application.

In other works like dOpenCL [Kegel et al., 2012], Hybrid OpenCL [Aoki et al., 2010], and SnuCL [Kim et al., 2012] the key idea is to merge the native OpenCL implementations in the nodes of a distributed system into a single meta platform of OpenCL, and their

implementation is built on top of a middleware that provide communications and eases the synchronization.

In dOpenCL [Kegel et al., 2012] is it possible to execute legacy OpenCL applications with minimal modifications to the source code. A major drawback of this approach is that implies an explicit management of all the devices in the system as well as manual balancing of the workload limiting the scalability and portability of the applications.

Hybrid OpenCL [Aoki et al., 2010] consists of a runtime system that provides an abstraction for different OpenCL implementations and a bridge program to connect the multiple OpenCL implementations over the network. In this work, the communication layer is implemented using sockets instead of the standard MPI middleware however, the employment of sockets can become too complex for large scale systems where the execution of collective operations must be as efficient as possible.

SnuCL [Kim et al., 2012] wants to provide a single system image for heterogeneous CPU/GPU clusters under the idea of a unified operating system with many devices attached to it. The API extensions and the runtime system provide collective communications and event synchronization services through a virtual shared memory approach. In spite of its advantages, the use of a centralized control to handle synchronization and messaging may lead to performance degradation for large scale clusters due to the intensive messaging between the compute nodes and the central control host. Another disadvantage is that this work lacks of scheduling algorithms delegating this complex issue to the programmer.

Sun [Sun et al., 2012] presented an extension to the OpenCL API with a set of functions designed to schedule computing tasks over CPUs and GPUs using pools of tasks. The scheduler employs a fixed policy and event dependencies to determine the next task to be executed. The selection of the device is delegated to the programmer and does not integrates any middleware for inter node communications limiting the scalability of the applications to only one node.

The clOpenCL [Alves et al., 2013] work is based on wrapper libraries with multiple daemons running on each node. The daemons are responsible for the management of its devices, and can redirect the execution of some tasks to other daemons in the cluster. The communication layer is built on top of Open-MX, a high-performance implementation of the Myrinet Express message-passing stack [Goglin, 2008]. This work do not integrates any scheduling tool forcing the programmer to register the devices on the daemons and to select which device must run the tasks.

In clMPI [Takizawa et al., 2013] the key idea is to provide wrapper libraries to ease the interchange of data between multiple GPUs located in different nodes. This work

employs advanced memory strategies like pinned memory allocations or pipelined communications to improve the performance of the applications, and avoids the coordination of blocking and non-blocking MPI calls to the programmer. In spite of the importance of performing efficient data transfers another important issues like task scheduling or device management are not addressed in this work.

The major disadvantages of the works implementing OpenCL extensions are:

1. Requires the use of APIs with overwhelming amounts of functions and full understanding of the OpenCL semantics.
2. Hiding the MPI interface might in turn limit the scalability of the applications because also hides other MPI functions like, distributed file sharing, one sided communications and collective routines which are fundamental for solving problems having big data sets.

3.2 Extensions to MPI

MPI (the Message Passing Interface) is a open standard for writing message passing programs based on the SPMD (*Single Program Multiple Data*) paradigm. MPI is considered as the *de facto* standard for distributed memory programming, and the 1.0 specification was first released in May 1994 [3] and included the definition of the basic terms for massage programming: point-to-point communications, collective operations, process groups, communication contexts, processes topologies, among others. A major update was presented in the second release of the specification in July 1997 that included dynamic process creation and management, one-sided communications, extended collective communications, external interfaces and parallel I/O.

In spite of their remarkable advantages two major concerns: thread safety and heterogeneous computing device integration. Although the MPI API was designed to be thread safe not all implementations can deal with multithreading requests. Aside of this the MPI specification don not includes any mechanism to cope with heterogeneous devices nor with their memory spaces thus requiring additional tools to complete the data transfers.

Some works found in the literature try to overcome the limitations of MPI extending its capabilities to be aware of memory objects located in heterogeneous devices. This is a promising approach given that the MPI semantics fits naturally into other

[3] Message Passing Interface Forum (Ed.) (1994, May). MPI: A Message-Passing Interface Standard. Retrieved September 18, 2016, from http://mpi-forum.org/docs/mpi-1.0/mpi-10.ps Version 1.0

non-distributed memory fields for example, in shared memory multicore processors, or computers with NUMA architecture.

In cudaMPI [Lawlor, 2009] as well as in many other CUDA-aware implementations of MPI like Open MPI, or MPICH2, the key idea is to provide the tools to achieve an efficient but transparent interchange of data between multiple GPUs. This kind of tools can exploit the advantages of high end network interfaces like myrinet or infiniband to improve the performance of the data transference as well as advanced memory management strategies like asynchronous copy, pinned memory allocations, etc. In spite of this advantages this kind of extensions do not integrate tools for handling the workload distribution problem and only bring support to exploit NVIDIA GPUs.

MPI-ACC [Aji et al., 2012] is a framework designed to allow end-to-end data transfer in accelerator based systems and is implemented on top of MPICH2. The key idea consists in keeping the MPICH2 runtime system aware of the existence of heterogeneous accelerators using specific function calls for those data transfers involving the participation of a GPU. Even though MPI-ACC provides efficient data transfers, the procedural programming model used there enforces the programmer to be aware of many other issues like device initialization and the scheduling problem.

Song [Song and Dongarra, 2012] presented a framework designed to ease the implementation of linear algebra applications where the data can be partitioned in blocks of arbitrary size without compromising the correctness of the algorithm. This framework first defines a static data distribution model based on the type of processing unit (CPU or GPU) and delegates the execution of tasks to the processing units over a specific block of data. The communications required to fulfill the data dependencies between tasks are performed automatically by the framework as well as the assignment of tasks to processing units which is made in order that minimizes the number of data transfers. The major problems with this framework are the assumptions made to perform a fair data distribution, i.e the assumption that all GPUs in the cluster have exactly the same capabilities.

The expectation that the data space can be divided in arbitrary sized and independent blocks is complex even for many applications in linear algebra for example, in matrix multiplication.

In general the current works extending the capabilities of MPI focus on simplify the data transfers between the accelerators but leaving aside other important issues like management and scheduling problem when dealing with multiple heterogeneous accelerators.

3.3 Compiler Directives and Code Annotations

The use of compiler directives (pragmas) and code annotations represent an important direction to ease programming and to perform code analysis prior to the execution of the applications. This approach represents an important alternative to solve the scheduling problem minimizing the data transfers.

The OmpSs+OpenCL proposal [Elangovan et al., 2014] is a task-based programming model that provides portability and flexibility for sequential codes while the performance is improved by the dynamic exploitation of task level parallelism. Tasks in OmpSs are annotated with data direction clauses that specify the data used by the tasks and how it will be used. This work is suitable for programming heterogeneous multi-core and many core architectures but lacks of a middleware to harness distributed systems.

CoreTSAR [Scogland et al., 2014] is a set of OpenMP extensions and libraries for offloading loops and multithread computations to heterogeneous computing devices. This work provides services to ensure memory consistency, task association, and task scheduling. The distribution of data and tasks is described using "*pragmas*" however, as in other works of this kind the framework it requires a fixed and relatively simple pattern of memory accesses by each thread, and a fixed number of tasks.

In spite of the advantages of those works none integrates a middleware to perform inter node communication or synchronization thus limiting the scope of the applications to a single node.

3.4 Frameworks

The StarPU framework [Augonnet et al., 2011] consists of a runtime system, a set of compiler directives and a *c* API designed to schedule tasks on multiple computing devices. In order to harness clusters of workstations, StarPU uses MPI for the transference of data on devices located on different nodes. The first step in this framework consists in register the memory buffers required by each task in the StarPU runtime system using a special buffer handler. The buffer handler is a data structure that includes the Id of the home node, the access mode (R, W, RW) and a pointer to the host data. The number of buffers assigned to each task is specified using another data structure called "*codelet*" which is also used to define the procedure that the task will perform and the type of device where is expected to run (CPU, GPU or CUDA capable GPU). StarPU employs a centralized dynamic scheduling strategy with multiple dynamic heuristics to distribute the tasks and execute them once their data dependencies have been fulfilled.

In spite of their advantages for heterogeneous programming StarPU still requires several non-intuitive functions, structures, handlers, and deep understanding of the OpenCL semantics for the development of applications, enforcing more training to exploit the capabilities of the system. Another problem is related with scheduling because this work only considers the use of dynamic scheduling strategies however if the user have certain knowledge about the number of tasks to be executed or the features of the system, the use of static scheduling strategies can improve the performance of the applications.

G-Charm [Vasudevan et al., 2013] is a framework for executing message driven applications on hybrid clusters with GPU accelerators. G-Charm is built on top of charm++ [Kale and Zheng, 2009], an object-oriented, asynchronous message passing, parallel programming model. The charm++ programming model is based on remote method invocations (RMI) performed on instances of a class called *chare*. Each *chare* is handled using an independent thread, and is able to communicate with other *chares* using specific objects called *Proxies*. To bring support for message passing applications charm++ incorporates the adaptive MPI layer (AMPI) that implements the MPI specification on top of charm++. To harness the advantages of GPUs the programmer can incorporate *chares* with GPU requirements that will be compiled using the NVIDIA *c* compiler (*nvcc*) and linked with the rest of the code in the *chare*. The load balancing in G-Charm is achieved by collecting all the requests in a list of pending tasks and distributing the instances of *chares* to the faster device. The *chares* with GPU requirements must be instantiated in nodes with GPUs.

This work performs dynamic load balancing and also supplies fault tolerance, however some of its disadvantages are that when the amount of communications among *chares* is large the serialization and deserialization process inherent in any RMI middleware can become a bottleneck. Another drawback is the lack of portability of the applications due to the use of specific tools from NVIDIA.

Virtual OpenCL (VCL) [Barak et al., 2010] is a wrapper for OpenCL that allows unmodified applications to transparently utilize many OpenCL devices in a single cluster as if all the devices were on a local computer. It is designed to use different OpenCL implementations offering a shared pool of devices to the users. The communication between nodes is provided by MOSIX, a cluster operating system designed to simulate shared memory on distributed computing systems. Given the nature of MOSIX, VCL can provide an extended API of OpenMP to ease the spawning of processes enabling also efficient collective operations. Among the major disadvantages of VCL are that it is not designed for portability due to the fact that requires a homogeneous Linux kernel

installed on each node. Another disadvantage is that the TCP/IP sockets used for communications are harder to use than the MPI programming interface and the scheduling problem is completely delegated to the programmer.

3.5 Summary

In Table 3.1 we summarize the major features of each work. Most of the works found in the literature are designed to simplify the development of applications hiding parts of the code required to perform data transfers or to manage the computing devices however, they do not provide support to complete all the steps in the life cycle of the applications from its development to its execution.

Programming Tools	Implementation Mode	Work Name	Middleware	Scheduling Technique	Programming Model
APIs & Runtime Systems	OpenCL Extensions	libWater	MPI	Dynamic	Tasks
		clMPI	MPI	None	Procedural
		dOpenCL	GCF	None	Procedural
		SnuCL	MPI	None	Procedural
		clOpenCL	Open-MX	None	Procedural
		Hybrid OpenCL	RPC	None	Procedural
	MPI Extensions	cudaMPI	MPI	None	Procedural
		Distri. GPUs	MPI	Dynamic	Tasks
		MPI-ACC	MPI	None	Procedural
	Frameworks	StarPU	MPI	Dynamic	Tasks
		VCL	MOSIX	None	Procedural
		Gcharm++	RMI	Dynamic	OOP
Compiler Directives	Pragmas and Code Annotations	OmpSs+OpenCL	None	Dynamic	Tasks
		CoreTSAR	None	Dynamic	Tasks

Table 3.1: Major features found in some of the related works

Part II

Contributions

Chapter 4

The XSCALA Framework: Architecture and Operation

The *Xplatform SCALAble* (XSCALA) is a framework designed to provide services and tools for the implementation and execution of task-based applications in H/H architectures. In this chapter we present the architecture and operation of the XSCALA framework.

4.1 XSCALA Framework

XSCALA is organized in three layers: Front-End, Middleware, and Back-End.

The **Front-End** provides the facilities for the development of the applications and the analysis of the source code. This layer is composed of XSCALA API and the static analyzer plugin.

The **Middleware** implements the services required for the execution and communication of tasks. This layer is composed of four modules with multiple modules attached to them.

The **Back-End** layer performs the interaction with the computing devices and works as an interface between the concept of tasks used in the middleware layer with the abstract object handlers used in OpenCL.

A global component view of the architecture of the XSCALA framework is depicted in the Figure: 4.1. Next we present the operation of the Front-End and Back-End layers and the explanation of the middleware architecture is deferred to chapter 5.

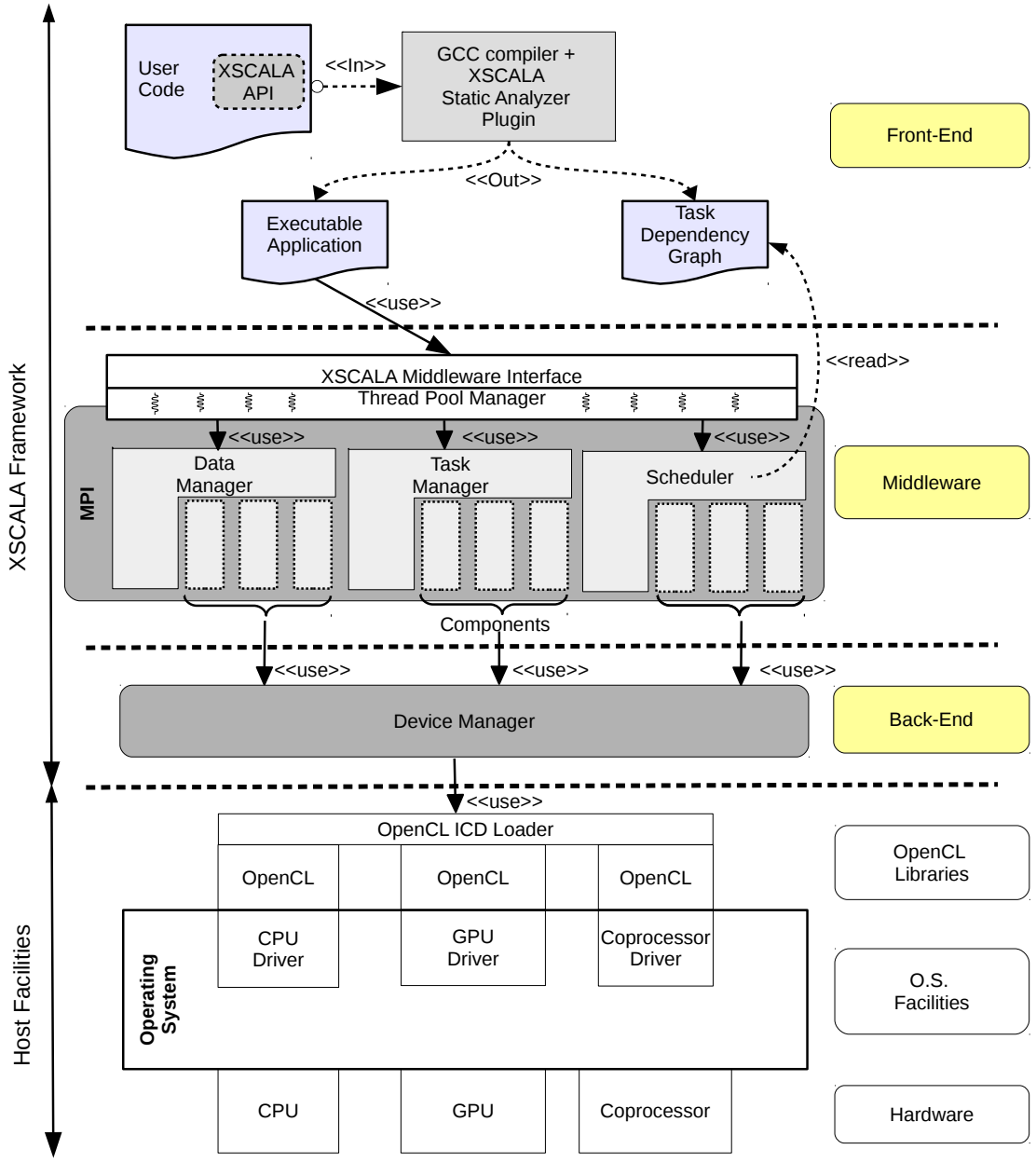


Figure 4.1: Architecture of the XSCALA framework. The solid arrows indicate the use of the interface to requests a function or service and the dashed arrows indicate the flow of files.

4.1.1 Front-End Layer

From a user perspective the framework works in two stages: code analysis and execution. The code analysis automates the construction of the task dependency graph used to improve the performance of scheduling algorithms, once the analysis is finished the compiler produces the executable file.

The Figure: 4.2 summarizes the operation of the code analyzer where the programmer

supplies its application code including functions of the XSCALA API. The code is compiled using our compiler, which has been extended with our static analyzer plugin, and two objects are created: the executable application and the task dependency graph. The details about the operation of the analyzer are presented in section 4.3.

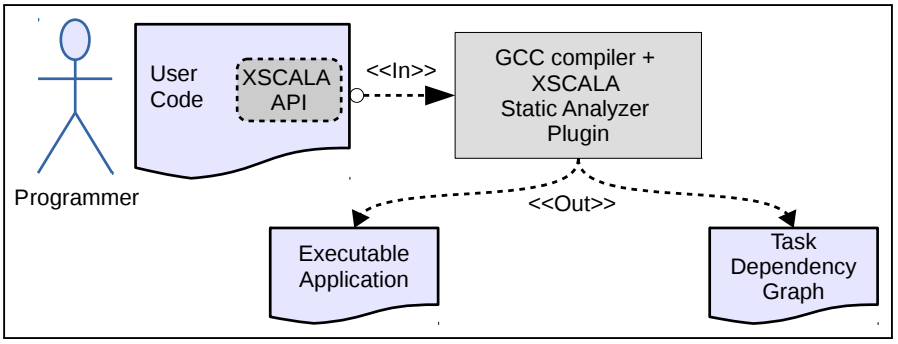


Figure 4.2: XSCALA Front-End Components.

4.1.2 Back-End Layer

The OpenCL API is a low level programming tool that requires several object handlers with cumbersome semantics to use the computing devices. The objective of the Back-End layer is to work as an interface between the requests of the tasks made in the middleware and the object handlers used in the OpenCL implementations. An object handler is a pointer to an abstract data structure used by the operating system and by the device driver to transfer data from host to device memory and conversely.

The *Back-End* layer is composed of a library called device management that implements the following functions: device exploration, device initialization and device filtering. The Figure 4.3 shows a component view with the interactions of the device manager layer with the OpenCL Instalable Client Driver Loader (ICD Loader).

The ICD is an extension of the specification that allows multiple implementations of OpenCL to co-exist on the same system. Each implementation of OpenCL is refereed to as a platform and works side by side with the driver of the hardware.

The OpenCL ICD Loader works as follows: The application first must query the number of platforms installed in the system and the ICD assigns returns a list setting a unique ID for each one of them. The ID allows the applications to choose which platform in the list wants to use to complete each one of the forthcoming OpenCL API calls.

The three functions implemented in the device management module work as follows:

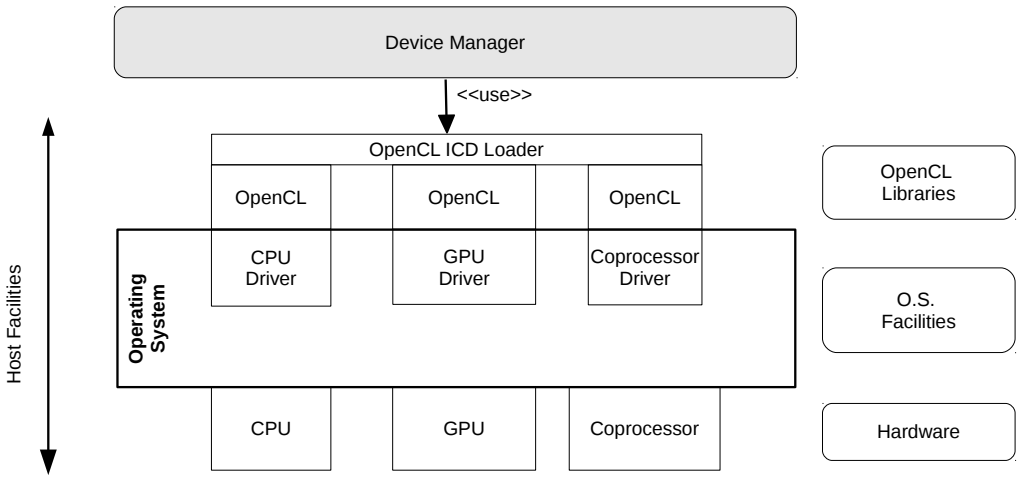


Figure 4.3: Components of the XSCALA Back-End layer.

Device Exploration

The objective of device exploration is to count the number of devices and query the properties of each one. The exploration is performed per node and works in two stages: First the device manager makes a request to the ICD loader to count the number of OpenCL platforms installed in the node, this query returns the number of platforms and a platform object handler for each implementation. The handler includes a unique ID for the platform called *PlatformID*. In the second stage the device manager queries the number and the properties of each device supported by each platform and again initializes a device object handler that includes a unique ID per device called *DeviceID*.

Device Initialization

Once the ID for the device is defined we need to initialize other two object handlers required to use the device: the context object and the queue object. The former is required to map the function calls to the appropriate platform and the latter is required to request the execution of subroutines in the device.

The Figure: 4.4 shows a hierarchical representation of all the handlers required to create and execute an application and works as follows: the first handler is the PlatformID, next the DeviceID is created using the PlatformID, next the Context is created using the DeviceID, and Finally the Queue is created using the Context object.

When the application wants to transfer data from the memory requires a buffer object that is constructed using the context created before next it must enqueue the operation in the queue of the device. Similarly to execute a kernel the program object must be

first and the kernel object is created using the program object finally the request to execute the kernel must be enqueued.

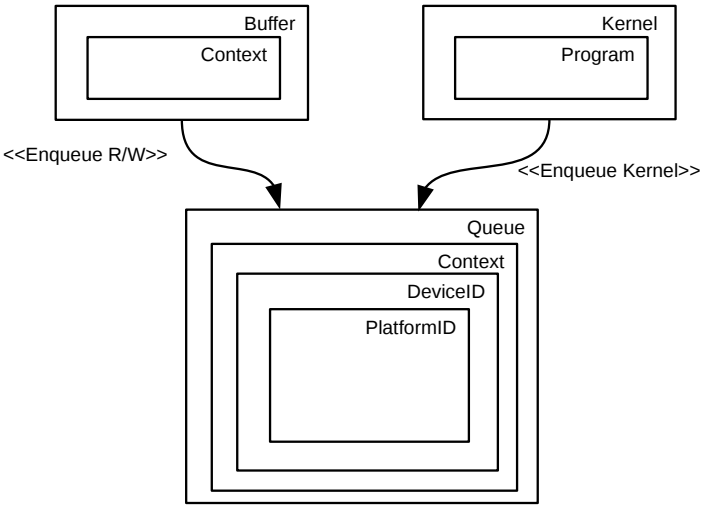


Figure 4.4: Object handlers required by OpenCL. The kernel execution and buffer operations must be sent to the device queue.

Device Filtering

Finally the device management layer implements the device filtering function. The objective of this function is to notify the middleware about the set of devices that meet a criteria for example, the type of device, the amount of memory or the max allocation size enable per device. This function is used by the scheduler to match the tasks with suitable devices.

4.1.3 Execution View

For the execution of an application, XSCALA starts a new process in the operating system of each node in the cluster and invokes the device exploration function in the Back-End layer which will notify the middleware about the number and properties of the devices found. Once the exploration has finished the control of the process is transferred from XSCALA to the application that hereinafter will invoke the XSCALA functions.

The Figure: 4.5 shows the components involved in the execution of the applications using up to m nodes having multiple processing units on each node.

The initialization of the XSCALA environment is always the first function invoked by the applications and is requested using the `XSCALA_Initialize` function call. This call triggers the following operations:

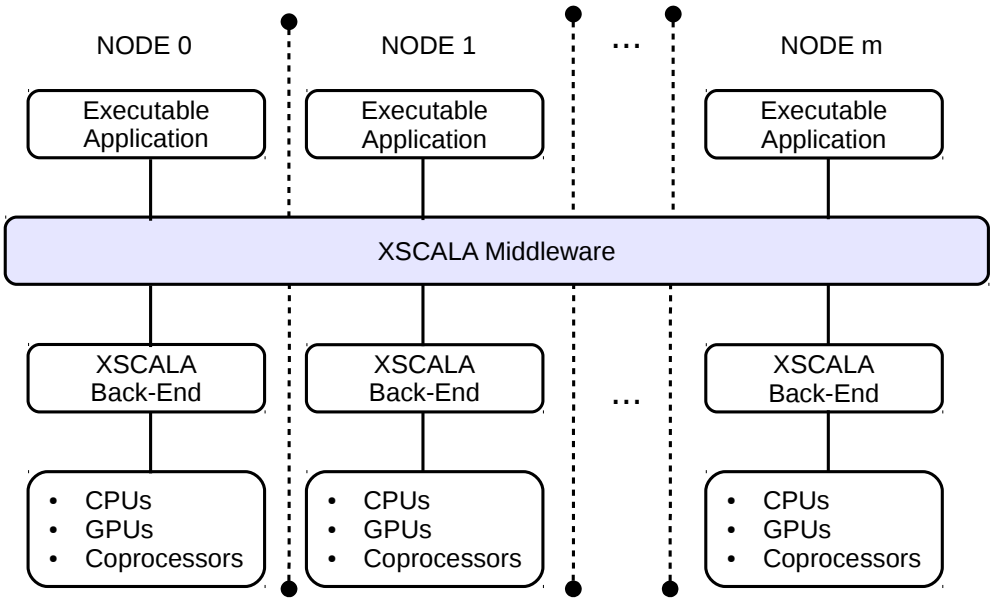


Figure 4.5: Application Execution View

(i) Parse the command line arguments searching the options specified by the user. The possible options are:

- `DEVICETYPE`. This option specifies the type of device that we want to use for example: `DEVICETYPE=CPU_ONLY`, `DEVICETYPE=GPU_ONLY` or `DEVICETYPE=ALL_DEVICES`.
- The scheduling mode. This option is composed of the mode and an optional parameter indicating the name of the heuristic to be used for example: `STATIC=HEFTMC` or `DYNAMIC=RR`. In case of manual scheduling we use the option `"TASKFILE"` including as parameter the path to the configuration file for example: `"TASKFILE=./Task/File/Path"`.
- `AUTOTUNE`. This option is used to request the execution of an autotuning process useful to improve the performance of the scheduling algorithms. This option is used without any parameter.

(ii) Request the device management library to filter and initialize those devices selected by the user.

(iii) Request the scheduler to plug the components for the scheduling mode specified by the user and build the mapping of task with computing devices. The scheduler can query the results of the autotuning process and the task dependency graph to complete its execution.

(iv) Request the thread pool to add a new thread for each task allocated in the node.

Once the initialization is finished the application can proceed to request more functions of the XSCALA API (see section 4.2.3, on page 51). The function calls are delegated to the corresponding task-thread that hereinafter is the responsible for executing the subroutines invoking the functions in the data management and task management modules.

When the application requires a data transfer the data management module finds the location of the source and the destination devices involved in the transference to automatically perform the memory allocation and the copy. If the transference involves a remote node the data manager uses the MPI services to complete the transference.

When the application requests the execution of a procedure the task-thread access the device handlers in the device manger and performs the compilation of the procedure. Next it invokes the functions to parse and sets the arguments of the procedure to the task manager module.

The finalization of the XSCALA environment must be the last request of the applications. This function sends a finalization signal to the thread pool manager which ensures that all pending operations are completed before the exit.

4.2 Programming with XSCALA

In the task programming model the programmer will focus on expressing its algorithms as collections of "*tasks*" with data dependencies among them, whereas, the selection of the computing device and the communications are delegated to the XSCALA middleware.

In this section we present the elements required to express the algorithms using our task approach. We first introduce the two basic structures: task and dependencies. Next we present the elements of a task and finally we present the XSCALA API.

4.2.1 Tasks

In the scope of this work a ***task*** is a structure constituted of three elements: the data, the procedure, and the worker device labeled with a unique identifier called the **taskID**.

A remarkable feature in the task programming model is that tasks are self-contained structures. The three elements of the task data structure are depicted in the Figure: 4.6 and are defined as follows:

The procedure or program, is the set of sequential instructions that must be executed.

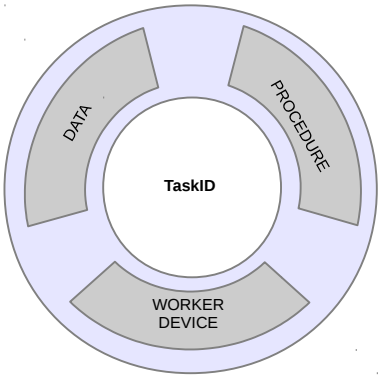


Figure 4.6: Basic components of a task.

The worker device is the processing unit the executes the procedure and is selected by the scheduling algorithms.

The data the element to be processed, is the information stored in the memory of the worker device. The data can be stored using data transfers or write operations however to keep a trade-off between throughput and ease of programming our framework introduces the concepts of *"Entities"* and *"Trays"* defined as follows:

i) Entities: are defined as indivisible blocks of data. The objective of the *"Entities"* is to enable the runtime system to split large data files between tasks requiring minimal user interaction but ensuring the coherence of the data.

ii) Trays: are defined as containers of the data entities assigned to one task. The objective of this concept is to simplify the management of data because trays are components tied to one tasks regardless of the devices where the task is allocated.

The creation of tasks can be carried out in two forms: at runtime using a explicit request or at compile time using configuration files. In both cases the elements of the task are coupled by XSCALA at runtime.

4.2.2 Data Dependencies

One of the most important responsibilities of any runtime system supporting task programming is to ensure compliance of data dependencies. A data dependency is a structure used to establish relations of order between the tasks *i.e.* which task goes before.

Data dependencies appear when a task must wait for other, called the predecessor tasks, to complete in order to retrieve some data from it for example, in the Figure: 4.7 the task B must wait for task A to complete before to retrieve the *data* and starts its execution. In this case we say that B has a dependency on A.

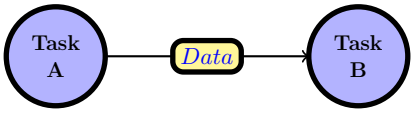


Figure 4.7: A data dependency example, in this case we say that B has a dependency on A

Once the data dependencies are met the execution of the task can be started but once it is started it must be completed without requiring neither data transfers nor synchronization with other tasks, as a consequence of this, all data transfers can occur after or before but never during the execution of the task. Based on this observation we establish that: **the data dependencies in XSCALA are implicitly declared in send-receive operations between tasks**. Send-receive operations declare the source and the destination of the data to be exchanged thus providing all the information required to establish the dependency.

4.2.3 The XSCALA API

The XSCALA framework includes an API with functions designed to the creation and management of tasks. These functions are classified into three groups: System Management, Task Management, and Data Management. All the XSCALA functions have a syntax resembling the MPI API that eases the learning process, however, the functions in our proposal are designed for the creation and the communication between tasks instead of processes.

System Management Functions

This set of functions is designed to make queries about the number of tasks in the system or the number and type of the computing devices and work as follows:

- `XSCALA_Initialize` initializes the XSCALA environment.
- `XSCALA_GetNumTasks` gets the number of tasks registered in the system and returns the result in the address pointed by `numTasks`.

- `XSCALA_GetNumDevices` gets the number of devices of a specific type (`CPU_ONLY` ,`GPU_ONLY`, `ALL_DEVICES`) registered in the system and returns the result in the address pointed by `numDevices`.
- `XSCALA_Finalize` finalizes the XSCALA environment.

The specific signature of these functions is shown in the Table: 4.1

`int XSCALA_Initialize(int argc, char** argv)`
`int XSCALA_GetNumTasks(int* numTasks , MPI_Comm comm)`
`int XSCALA_GetNumDevices(int* numDevices, int DEVICE_TYPE, MPI_Comm comm)`
`int XSCALA_Finalize()`

Table 4.1: System Management Functions

Data management functions

Data management functions are designed for transferring data between different tasks and works as follows:

- `XSCALA_Scatter` distributes the contents of a data file between all the tasks.
- `XSCALA_Gather` collects the contents of a specific tray of all tasks in a single data file.
- `XSCALA_SendRecv` enables the transference of data between two tasks regardless of their device.
- `XSCALA_ReadTray` transfers the contents of a tray in one task and stores to the specified space in host memory .
- `XSCALA_WriteTray` transfers the contents of the specified host memory space in one tray of one task.
- `XSCALA_CommitEntity` defines the entity type.
- `XSCALA_Reduce` takes as input the source tray the destination tray and the root task, to perform the requested operation for each entity of each source tray and store the results in the destination tray.
- `XSCALA_AllReduce` performs the same operation that in `Reduce` but keeping a copy of the results on the destination tray of each task.

- `XSCALA_MallocTray` reserves memory in the device where the task is allocated.
- `XSCALA_FreeTray` free the memory spcace used by the tray in the device where the task is allocated.
- `XSCALA_FreeAllTrays` free the memory spcace used by all the trays in the device where the task is allocated.

The signature of the functions in this group are shown in Table: 4.2.

`int XSCALA_Scatter(const char* datafileName, MPI_Datatype entityType, int trayId, MPI_Comm comm)`
`int XSCALA_Gather(int trayId, MPI_Datatype entityType, const char* datafileName, MPI_Comm comm)`
`int XSCALA_SendRecv(int src_task, int src_trayId, int dest_task, int dest_trayId, long int traySize, int TAG)`
`int XSCALA_ReadTray(int taskId, int trayId, long int traySize, void * hostBuffer, MPI_Comm comm)`
`int XSCALA_WriteTray(int taskId, int trayId, long int traySize, void * hostBuffer, MPI_Comm comm)`
`int XSCALA_CommitEntity(int blockcount, int* blocklen, MPI_Aint* displacements, MPI_Datatype* basictypes, MPI_Datatype * entityType);`
`int XSCALA_Bcast(int rootTaskId, int trayId, long int traySize, MPI_Comm comm)`
`int XSCALA_Reduce(int rootTaskId, int src_trayId, int dst_trayId, long int traySize, MPI_Datatype entityType, int OPERATION, MPI_Comm comm)`
`int XSCALA_AllReduce(int src_trayId, int dst_trayId, long int traySize, MPI_Datatype entityType, int OPERATION, MPI_Comm comm)`
`int XSCALA_MallocTray(int taskId, int trayId, long int traySize, MPI_Comm comm)`
`int XSCALA_FreeTray(int taskId, int trayId, MPI_Comm comm)`
`int XSCALA_FreeAllTrays(int taskId, MPI_Comm comm)`

Table 4.2: Data Management Functions

Task management functions

Task management functions implements the basic operations of our task programming model, they are designed to create, execute, and free the tasks, as well as for the establishment of synchronization points. These functions work as follows:

- `XSCALA_CreateNewTask` is used to create new tasks at runtime. This function performs the register of the tasks in the system using a distributed mutex algorithm to update the global address table (see section 5.5, on page 74) avoiding data races in the assignment of IDs to the new tasks.
- `XSCALA_SetProcedure` is used to establish the procedure in the specified task.
- `XSCALA_ExecTask` requests the execution of the procedure.
- `XSCALA_FreeTask` releases all the resources used by the specified task.
- `XSCALA_FreeAllTasks` releases all the resources used by all the task registered in the system.
- `XSCALA_WaitFor` sets a synchronization point between the task involved in the call. When one task reaches the synchronization call no more work can be fetched until all other tasks reach the synchronization point.
- `XSCALA_WaitAllTasks` sets a synchronization point between all task registered in the system. When the task reaches the synchronization call no more work can be fetched until all other tasks reach the synchronization point.

The signature of the functions in this group are shown in Table: 4.3.

`int XSCALA_CreateNewTask(task_t * task, int numTasks, MPI_Comm comm)`
`int XSCALA_SetProcedure(MPI_Comm comm, int TaskId, const char * srcPath, const char * procedureName)`
`int XSCALA_ExecTask(MPI_Comm comm, int taskId, int workDims, size_t * globalThreads, size_t * localThreads, const char * fmt, ...)`
`int XSCALA_FreeTask(int taskId, MPI_Comm comm)`
`int XSCALA_FreeAllTasks(MPI_Comm comm)`
`int XSCALA_WaitFor(int numTasks, int* taskId, MPI_Comm comm)`
`int XSCALA_WaitAllTasks(MPI_Comm comm)`

Table 4.3: Task Management Functions

4.3 Static Code Analysis

The objective of the static code analyzer consists in automate the construction of the task dependency graph of the application required to improve the performance of static scheduling algorithms.

Our code analyzer tool is implemented as a plugin for the *c* compiler "gcc" that makes an exploration of the Abstract Syntax Tree (AST) of the code before the compilation of the application. The AST of the applications is created using the Low Level Virtual Machine (LLVM) infrastructure and clang as front end *c* compiler (LLVM/clang).

LLVM is a collection of modular and reusable compiler and toolchain technologies [Lattner, 2008] useful to extend the capabilities of traditional compilers. Our plugin finds the declarations of tasks and the data dependencies among them achieving an automatic construction of the task dependency graph.

To perform this search we assume that the applications have the structure depicted in Listing: 4.1 that works as follows: In the line 5 we have the call to *initialize* the XSCALA environment and plug the components in the system. In the line 6 we have a declaration of a single task and in Line 7 we declare an array of two tasks.

Here is worth to say that the initialization step commented out in line 8 is required only for dynamic scheduling for cases when more tasks need to be added at runtime. Similarly the task declaration of lines 6 and 7 can be omitted in case of manual scheduling where the number of tasks is declared in the configuration file. In those cases the code analyzer becomes useless.

Once the tasks are declared we proceed with the *body of the application* for example: in line 10 we allocate the `TRAY_0` for the task `myTasks[0]` and in line 12 we write data into this tray. In line 13 we set the procedure of the task. In line 14 we request the execution of the procedure passing the parameters to the procedure using a list of arguments like in a `printf` call, the details in the format of the parameter list are described in section 5.3.2. In line 16 we perform a `SendRecv` operation that establishes a data dependency of tasks `myTasks[1]` on `myTasks[0]`. Finally in line 21 we request the *finalization* of all pending tasks to enable the release of resources.

In the Figure: 4.8 we show a section of the AST generated from our sample code. To get the number of tasks declared in the code we must match all the nodes in the AST labeled as "*VarDecl*" and with the data type "`task_t`". In this sample we match two nodes with names *mySingleTask* and *myTasks* that are related with the declarations made in the line 6 and 7 of our sample code.

Similarly, the data dependencies can be determined by looking into the parameters of the "`XSCALA_SendRecv`" function calls. Here is worthwhile to say that when the parameters are provided as dynamic variables our analyzer becomes unable to determine the data dependencies.

```
1  #include XSCALA.h
2  #define TRAY_0 0
3  #define SIZE 128 // size of the tray in Bytes
4  /*========Initialization==========*/
5  XSCALA_Initialize(int argc, char*[] argv);
6  task_t mysingleTask;
7  task_t myTasks[2]; //An array of two tasks
8  // XSCALA_CreateNewTasks(myTasks, 2, MPI_comm); //Initialization of tasks
9  /*=====Body of the Application=====*/
10 err= XSCALA_MallocTray(myTasks[0].ID, TRAY_0, SIZE, MPI_comm); //Tray initialization
11 err|= XSCALA_MallocTray(myTasks[1].ID, TRAY_0, SIZE, MPI_comm);
12 err|= XSCALA_WriteTray(myTasks[0].ID, TRAY_0, SIZE, hostBuffer, MPI_comm);
13 err|= XSCALA_SetProcedure(MPI_comm,  myTasks[0].ID, "Path", "kernelName");
14 err|= XSCALA_ExecTask(MPI_comm, myTasks[0].ID, WORKDIMS,
15              globalThreads, localThreads, "%d,%d", ...); //Execution request
16 err|= XSCALA_SendRecv(myTasks[0].ID, TRAY_0, myTasks[1].ID, TRAY_0, SIZE, TAG);
17
18          . "More lines of code"
19
20 /*========Finalization==========*/
21 XSCALA_Finilize();
```

Listing 4.1: The structure of a program in XSCALA using dynamic task creation.

The Figure: 4.8 also includes the node "*DeclRefExpr*" that is related with the call to the "`XSCALA_SendRecv`" function in the line 16 of our sample code.

The parent node of the node have the parameters of the call with four of them provided as integers given directly in the "*IntegerLiteral*" nodes indicating the source and the destination trays of the data dependency, as well as the size of the transfer, however the source and destination tasks are presented in a different form due to the use of the arrays.

When the "`XSCALA_SendRecv`" function call is performed using names of variables or arrays in its parameters we need to find the element of the array being referenced making a deep exploration of the call. The Figure: 4.9 we shows another section of the AST of the code, there we found a reference to the element 0 of an array of type "`task_t`" called "`myTasks`" and the top leafs in the tree indicates an acces to the ID member followed by a cast to the type of the function prototype,

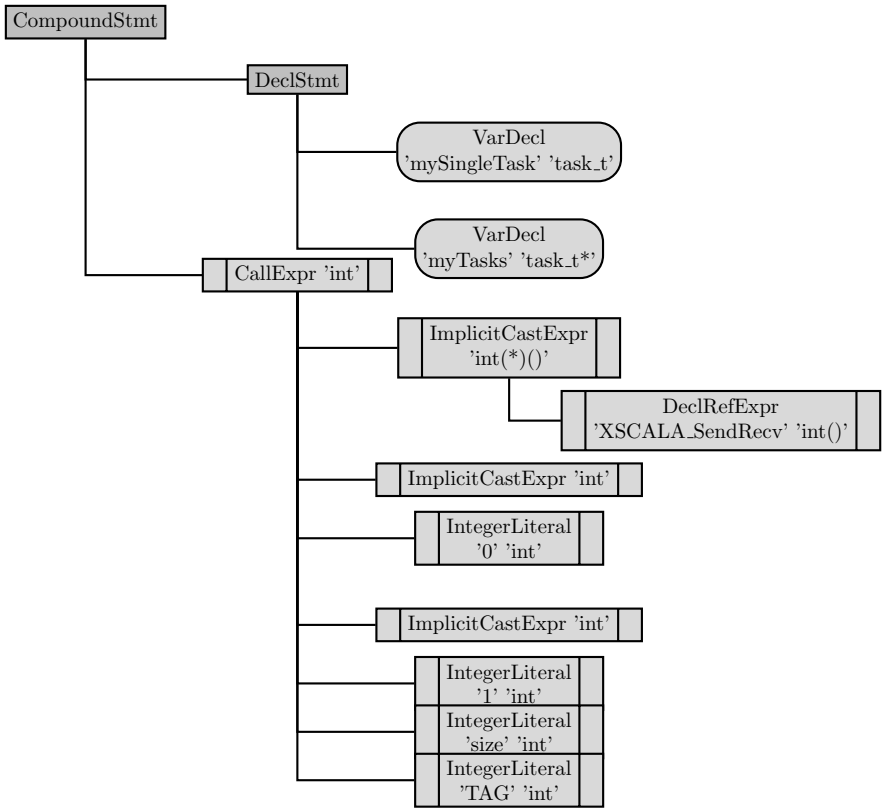


Figure 4.8: Exploration of an AST to match the `XSCALA_SendRecv` function call. The data dependencies are established using the parameters found in this function call

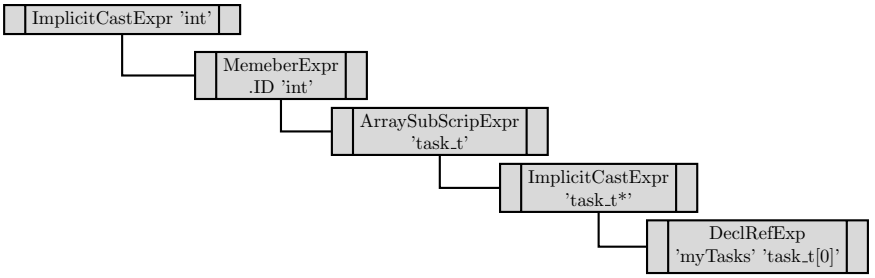


Figure 4.9: Exploration of the AST to find out source task. In this example we matched the reference of the first element of an array called "`myTasks`".

Chapter 5

XSCALA Middleware Architecture

The XSCALA middleware provides the services for the execution, synchronization and communication between tasks. In this chapter we first present the architecture of the middleware including a description of each module and of the algorithms that they implement next we present a formalization of the execution model and a brief discussion of the correctness of the order on which the tasks are executed.

5.1 Architecture

The middleware consists of four modules: the thread pool manager, the data manger, the task manager, and the scheduler. Each module might have several components attached to it as is depicted in the Figure: 5.1. The components provide the implementation of the algorithms required to complete the application and to ensure the correctness during the execution. The objective of the modular component architecture is to enable the substitution or extension of individual components in the implementation to improve the performance of the applications.

The objective of the modules is to enable the transition between the states in the lifecycle of a task for example, the register in the system, the execution of the procedure and its final release. Our middleware is built on top of OpenMPI [Gabriel et al., 2004b], an open source implementation of MPI, which provides inter node communication but is extended with the modules pointed out before.

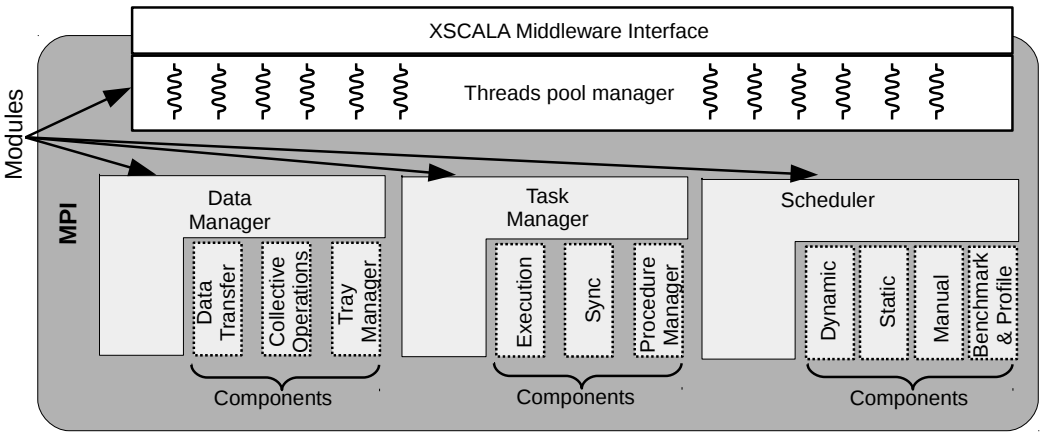


Figure 5.1: Internal architecture of the middleware layer. Each module have several components that implements specific functions.

5.2 The Threads Pool Manager Module

To maximize the performance of any application we must try to execute multiple task in parallel and overlap computation with communications as much as possible. The objective of the threads pool manager is parallelize the execution of multiple tasks using multiple devices and perform data transfers as soon as possible, respecting the precedence relations.

To achieve this goal this module creates a pool of "*task-threads*" to whom the *"process-thread"* can delegate the execution of small subroutines. Each task-thread is thus responsible for complete the execution of all the subroutines related with its task. We refer to the thread executing the application on the node as **process-thread** and the threads performing the subroutines associated with the task as **task-threads**.

An overview of the components related with the threads pool manager is depicted in the Figure: 5.2. In this figure a process is running in one node of the system with multiple task-threads performing the subroutines delegated by the process-thread, with each task allocated in a different device.

5.2.1 Task Delegation

In order to avoid wasting CPU cycles with each task-thread asking for more work, the thread pool module implements the producer-consumer pattern [Schmidt et al., 2000] using the process-thread as the producer, and the task-threads as the consumers with a unbounded buffer of subroutines. The operation of this pattern is depicted in the sequence diagram of Figure: 5.3.

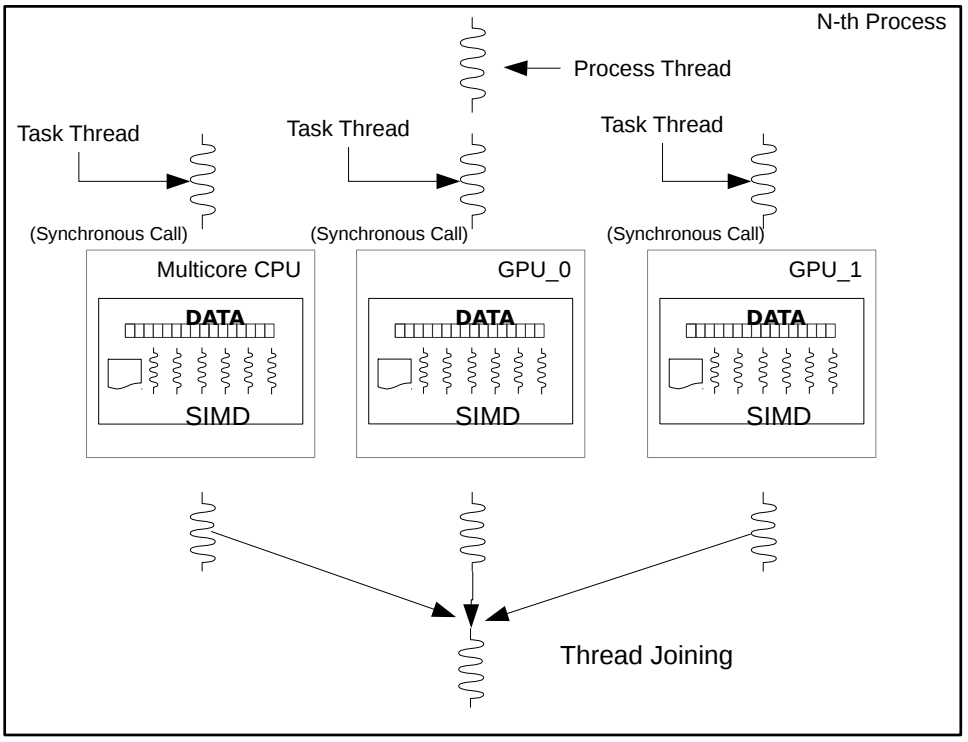


Figure 5.2: Parallel execution of multiple tasks on multiple devices. The process-thread spawns multiple independent threads to complete the execution of the tasks.

In this case the buffer consists in a list of subroutines and their parameters that are read by the process-thread and delegated to the task-thread for example, compile the kernel receive data, send data, etc.

The thread pool manager is responsible for coordinate the integration of new tasks to the pool at runtime *i.e.* enables the dynamic creation of tasks. This is a complex operation that must avoid data races at the time of inserting new buffers in the pool therefore it requires the coordination between all task-threads.

To ensure a correct integration of threads the producer consumer pattern is combined with the readers writers pattern [Courtois et al., 1971] with the objective of ensuring that current task-threads wait while the new buffers are included and avoid data races due to the possible reallocation of the current work buffers.

The Algorithm: 1 in page 63, implements the patterns aforementioned and works as follows: In lines 2 to 5 the task subscribes as the reader, in line 6 the task checks if his buffer has pending work, in lines 7 to 9 the task checks if it can exit, the locks in lines 10 to 12 lets the task fetch a subroutine from its buffer without data races. The lines 14 to 19 implement the protocol executed by the process-thread to insert a new buffer in line 15 requests the task to leave and in line 17 performs the reallocation of the buffers. The *delegateWork* procedure is performed by the process-thread when delegates work

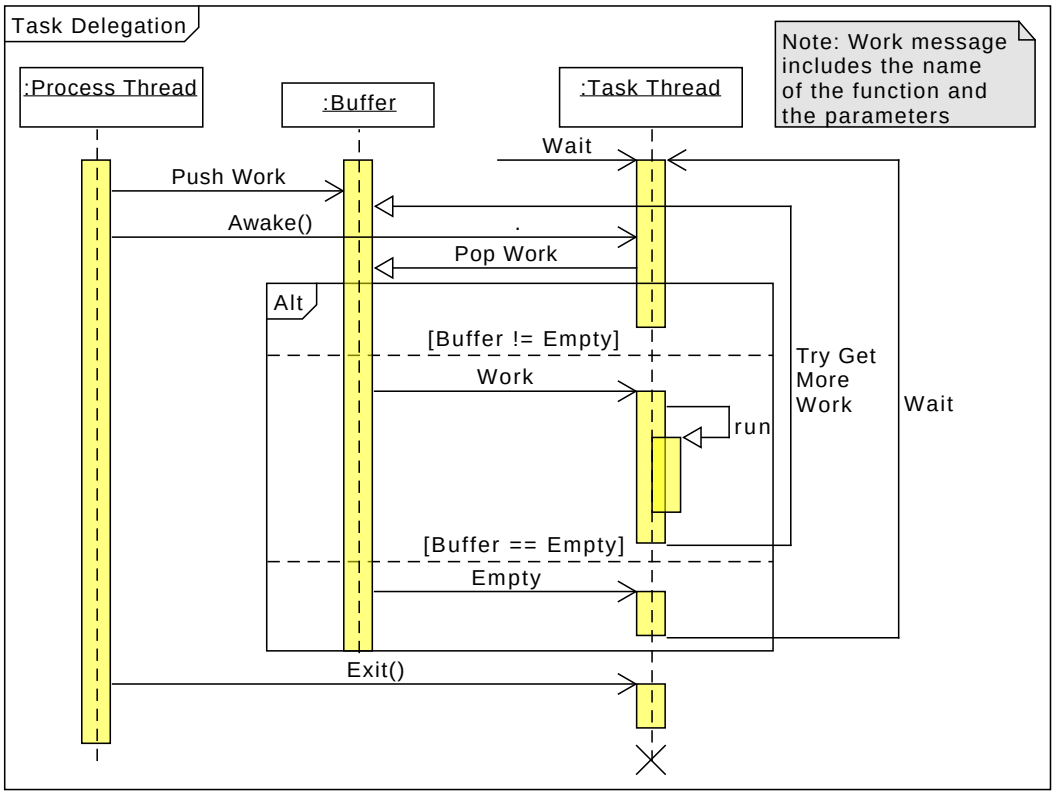


Figure 5.3: Sequence diagram of the task delegation process. The process-thread delegates subroutines to multiple task-threads through the work buffer.

to one task, while the *readerSubscribe* and *readersUnsubscribe* procedures are executed by the tasks when they want get access to its buffer or are requested to wait.

5.2.2 Dynamic Task Integration

One issue arising in dynamic task integration is that the new task-thread must be attached to the process that holds the device indicated by the dynamic scheduler algorithm, however this decision is delayed until the very last moment of requesting the execution of the task since in that moment we known all the data dependencies of the task. As a consequence of this, the task-thread might not be ready in the pool of tasks when the process-thread tries to delegate its subroutines.

To overcome this problem the thread pool manager implements a list called *callsBag*. The objective of the *callsBag* is to temporarily store all the subroutines that must be delegated to the task-thread until the selection of the device and the integration into the pool has been completed, at this moment all the subroutines stored in the list are transferred to the buffer of the task.

```
mutex myBufferMtx, readTry, arrayOfBuffers, readersContention;
semaphore myBufferSemp;
```

```
     /*Task thread (reader)*/
 1: while TRUE do
 2:    if myStatus!=Subscribed  then
 3:       readerSubscribe()
 4:       myStatus=Subscribed
 5:    end if
 6:    wait(myBufferSemp) // if there is no work in the buffer then sleep
 7:    if  (exitSignal AND noMoreWork)  then
 8:       EXIT
 9:    end if
10:    lock(myBufferMtx)
11:    pop(myWorkBuffer)
12:    unlock(myBufferMtx)
13: end while
    /*Process thread (writer)*/
14: lock(readTry)
15: delegateWork(readersUnsubscribe()) //Stop all readers
16: lock(arrayOfBuffers)
17: ReallocateBuffers() //Here secure buffer insertions can be performed
18: unlock(readTry)
19: unlock(arrayOfBuffers)
```

Procedure delegateWork(subroutine)

```
1: lock(taskBufferMtx)
2: push(subroutine)
3: unlock(taskBufferMtx)
4: signal(taskBufferSemp)
```

Procedure readerSubscribe()

```
1: lock(readTry) //try to read
2: lock(readersContention) // only one reader at time
3: readersCount++
4: if readersCount==1 then
5:    lock(arrayOfBuffers) //No modification to the buffer is allowed
6: end if
7: unlock(readersContention)
8: unlock(readTry)
```

Procedure readersUnsubscribe()

```
1: lock(readersContention) // only one reader at time
2: readersCount - -
3: if readersCount==0 then
4:    unlock(arrayOfBuffers)
5: end if
6: myStatus=UnSubscribed
7: unlock(readersContention)
```

Algorithm 1: XSCALA's algorithm for the consumption of subroutines and integration of new tasks implemented in the pool manager module.

The pool of threads enables an improvement in the performance of the applications due to its ability to delegate multiple synchronous functions without blocking other independent tasks for example, it enables to overlapping synchronous computation in one task with asynchronous communications between other tasks ensuring the fulfillment of the data dependencies.

5.3 The Task Management Module

The task management module is the responsible for controlling the execution of the tasks *i.e.* for compiling the procedure, its execution and synchronization. This module has three components: the procedure manager, the execution component, and the synchronization component. Next we describe the main activities of each component and in the case of the synchronization component we present the algorithm used to implement a synchronization point.

5.3.1 Procedure Management Component

As we pointed out before the procedure of the task can be executed by multiple internal threads to reduce the time required for its completion.

Given that the instruction set of each device might be specific for certain type of architectures the procedure must be provided as a plain text object written in *OpenCL C*, a restricted version of the C99 language specification[4] indicating the set of sequential instructions that each thread will perform and using the `__kernel__` entry point as is described in the OpenCL specification[5] . The procedure will be automatically compiled at runtime by the procedure management component as follows:

The first step consists in getting the device where the task was allocated, this information can be retrieved from the address table stored in the scheduler (see section 5.5, on page 74). Then the component performs the construction of a program object, an abstract structure used to compile the text into machine code suitable for the architecture of the device. If an error occurs during this process it is reported to the user. The final step consists in creating the kernel object, another abstract structure used to request the execution of the procedure, the kernel object is in fact the handler object used by XSCALA to access the procedure of the task once that it was compiled and allocated in the memory of the device. At this moment the procedure is ready to be executed.

[4]ISO IEC (Ed.) (1999, December). Programming Language - C Retrieved September 18, 2016, from http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[5]The Khronos Group (Ed.) (2012, November). The OpenCL specification Retrieved September 18, 2016, from https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf Version 1.2

5.3.2 Task Execution Component

This component is responsible for performing the execution of the procedure of the task. The first step in this process consists in parsing the arguments in the function call to set them into the procedure. The arguments for the procedure are passed in a string following the prototype `%specifier,%specifier,...,%specifier` with the specifiers described in Table: 5.1.

Specifier	Input Type
d	integer
u	unsigned int
f	decimal floating point
c	char
s	string of chars
T	unsigned int
%	A "%" followed by another "%" character skips the former

Table 5.1: Specifiers for the parameters of the procedure.

Where `T` is an special specifier used in XSCALA and consists in a unsigned integer used to indicate the ID of the tray to be passed to the procedure of the task. Once the arguments have been parsed the execution component maps the trays into the memory objects allocated by the data management module and requests the execution of the procedure using the handler created by the procedure management component.

5.3.3 The Synchronization Component

The synchronization component is the responsible for ensuring that all the tasks involved in a synchronization operation reach a specific barrier and stay there while the other tasks arrive to the same point. Synchronization points also are used when the process-thread wants to accesses the memory of one tasks and the synchronization point ensures that the task has finalized all the operations before grant the access to the data.

Given that multiple task can be allocated in remote nodes, our approach to achieve an efficient synchronization requires two phases: First the process-thread sets a local barrier for the task-threads and waits for its completion. In the second phase all the process-threads synchronize using MPI primitives to ensure the completion of the synchronization point in remote nodes. Using this approach we can ensure that those task not involved in the synchronization point continue working without interruptions.

The synchronization protocol is shown in the Algorithm: 2 and works as follows: The lock in line 2 lets the process-thread to ensure that no task can signal it before it start to listen, in lines 3 to 5 delegates the *SynchSubroutine* to the tasks, the semaphore in line 6 waits all task to reach this point the line 7 performs the synchronization with the other nodes and lines 8 and 9 resume the execution of the tasks. The *SynchSubroutine* works as follows: the thread barrier in line 1 performs the task synchronization, the lock in line 2 ensures that the process is listening the response of the tasks and lets one task at time try to weak up the process-thread. The conditional wait in line 7 waits until the conclusion of the global synchronization operation.

mutex synchMutex;
condition condN, backCond;

/*On process-thread */

```
1: postBackCount=0
2: lock(synchMutex) //Ensure no task-thread can signal me before I start to listen
3: for each task-thread do
4:    push(SynchSubroutine)
5: end for
6: cond_wait(condN,synchMutex)
7: MPI_Barrier //distributed barrier
8: cond_broadcast(backCond) // Now wake up sleeping task-threads
9: unlock(synchMutex)
```

Procedure SynchSubroutine
/*On task-threads*/

```
1: Thread_Barrier //local barrier
2: lock(synchMutex) // Wait until process-thread is listening
3: postBackCount++
4: if postBackCount== |Tasks| then
5:    cond_signal(condN)
6: end if
7: cond_wait(backCond,synchMutex) // now wait MPI processes to synch
8: unlock(synchMutex) //Done continue working
```

Algorithm 2: Algorithm for synchronization points implemented in the task management module

5.4 The Data Management Module

The data management module have four components responsible for the memory allocations and for completing the data copy operations between pairs of tasks. In order

to improve the performance of the applications the copy strategy must be selected depending on whether both the source and the destination tasks are allocated in the same device (**intradevice copy**), in the same node but in different devices (**interdevice copy**) or in different nodes (**internode copy**). Next we review each strategy in detail.

5.4.1 Data Copy Component

Transferring data through several hierarchies in remote and heterogeneous devices is a critical step and the strategy used for transferring the data can impact in the performance of the application. The data management module is the responsible for complete the transfers as efficiently as possible.

In order to overlap computation with communications the data copy component must deal with the problem of synchronization between the sender and the receiver tasks *i.e.* it must ensures that each transference can be completed in asynchronous form to the sender regardless the time at which the receiver is ready to receive. Next we describe three copy strategies and how they solve the synchronization problem.

Intradevice copy

Intradevice is selected when both the source and the destination tasks are allocated in the same device. To synchronize the transfer we developed an intradevice copy algorithm based on the publisher subscriber pattern [Oki et al., 1994]. The idea behind our implementation of this pattern consists in let each sender publish notifications tagged with the ID of the receiver, here we call *"container"* to the data structure storing all the notification published.

When the receiver arrives it checks if there exists a container having its ID and if it is found the receiver can proceed to consume the container, otherwise waits until a new publication appears. This procedure is shown in the Algorithm: 3 and works as follows: The lock in line 1 avoid races with the subscribers, line 2 publishes the new container in the list and the broadcast in line 3 informs the subscribers that there are new publications. The wait in line 6 ensures that the subscribers wait for at least one publication, the lock in line 7 ensures only one task checks if there is new content of its interest if true consumes the container or if is false sleeps until a new publication appears letting other task to check the current publications avoiding starvation.

```
mutex synchMutex
condition condN
semaphore FULL
```

```
      /*On sender task*/
 1: lock(synchMutex) /*sender try to publish data*/
 2: pushNewContainer(TAG=receiverID)
 3: cond_broadcast(condN,synchMutex)
 4: signal(FULL)
 5: unlock(synchMutex)

     /*On receiver task*/
 6: wait(FULL) /*Wait for content in the container*/
 7: lock(synchMutex) /*Only one receiver enters to check the content */
 8: while TRUE do
 9:    Container=pop (DataContainer where TAG = myID)
10:    if  Container != NULL then
11:       CopyData
12:       unlock(synchMutex)
13:       exit
14:    else
15:       signal(FULL) /*Could not find my container*/
16:       cond_wait(condN,synchMutex) /*sleep to avoid starvation in other tasks*/
17:    end if
18: end while
```

Algorithm 3: XSCALA's algorithm for intradevice data copy

Interdevice copy

The interdevice copy is used when the source and the destination tasks are in the same node but in different devices. This copy is slower than intradevice copy because it requires an intermediate copy of data to the host memory as is depicted in Figure: 5.4.

Slight modifications to the algorithm 3 are required to achieve synchronization in iterdevice copy, such modifications are shown in the Algorithm: 4 and works as follows: In line 2 we append a *ReadData* operation to move data from device memory to local host memory (step 1 in figure 5.4), then in line 3 we publish a new container that includes a pointer to the data. Finally in line 12 we replace the copy operation with a write operation (step 2 in figure 5.4).

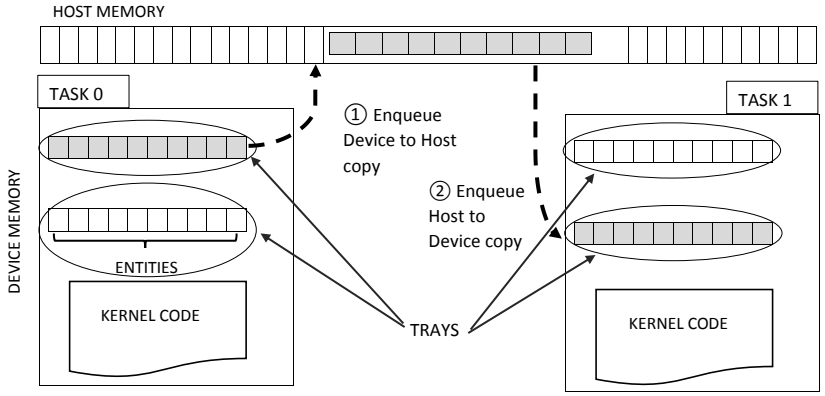


Figure 5.4: The interdevice data copy. This process executes a copy of data from the source device to the host, then makes a data transference between the two hosts using MPI and finally performs a copy of the data to the target device.

mutex synchMutex
condition condN
semaphore FULL

```
    /*On sender task*/
1:  lock(synchMutex) /*sender try to publish data*/
2:  ReadData
3:  pushNewContainer(TAG=receiverID, DataPointer)
4:  cond_broadcast(condN,synchMutex)
5:  signal(FULL)
6:  unlock(synchMutex)

    /*On receiver task*/
7:  wait(FULL) /*Wait for content in the container*/
8:  lock(synchMutex) /*Only one receiver enters to check the content */
9:  while TRUE do
10:   Container=pop (DataContainer where TAG = myID)
11:   if  Container != NULL then
12:     WriteData
13:     unlock(synchMutex)
14:     exit
15:   else
16:     signal(FULL) /*Could not find my container*/
17:     cond_wait(condN,synchMutex) /*sleep to avoid starvation in other tasks*/
18:   end if
19: end while
```

Algorithm 4: XSCALA's algorithm for interdevice data copy

Internode copy

Internode copy is selected when the source and destination tasks are allocated in different nodes, the Figure: 5.5 shows the internode copy process. Internode copy is the worst

case scenario in data transfers because it requires a copy of data from the memory of the source task to the memory of the host (step 1), then a MPI transfer to move the data from source to receiver host (steps 2 and 3), and finally the data is transferred to the memory of the destination device (step 4).

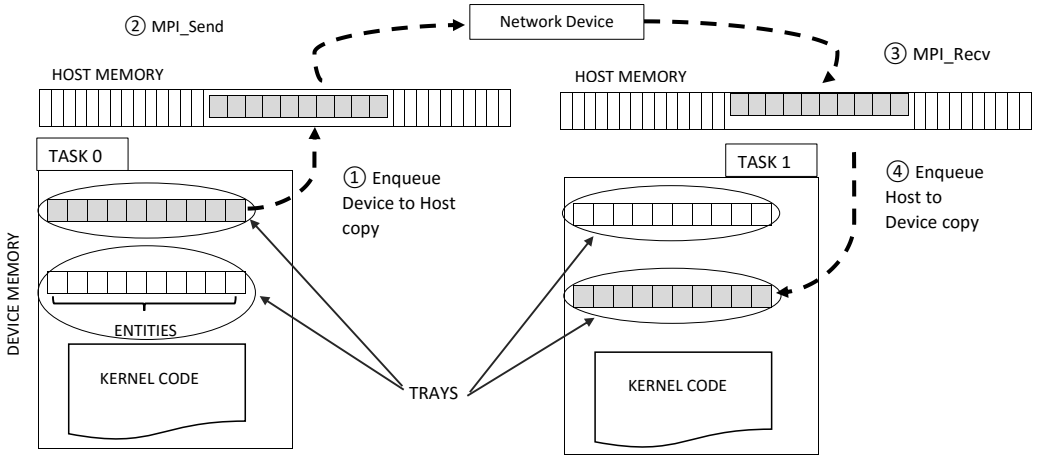


Figure 5.5: Internode data copy. This process executes a copy of data from the source device into the host, then makes a data transference between the two hosts using MPI and finally performs a copy of the data to the target device.

As in the last two cases internode copy must be aware of synchronization, however a remarkable problem with the message passing middleware is that it do not have support for multithreading. The MPI middleware was designed to work with multiple processes not with threads therefore we must solve the possible deadlock arising when two or more tasks try to access the interprocess communication socket in write mode. This conflict is depicted in Figure: 5.6 when task S1 try to reach task R1 and task S2 tries to reach task R2.

To overcome this limitation we add an intermediate thread called "*broker-thread*", and a new listener socket to each MPI process to complete the receive part of the communications. Using this approach we created the internode communication algorithm shown in the Algorithm: 5 in page 72 that solves the deadlock problem.

Our internode data copy synchronization algorithm is based in the MCS distributed mutex algorithm [Mellor-Crummey and Scott, 1991] and is combined with a distributed semaphore that can be implemented using on one sided communications to avoid the deadlock problem and works as follows: The lock in the line 1 ensures that one local task sends data at a time, and the MCS lock in line 2 ensures that there is no other remote task trying to sending data. Once both locks are obtained line 3 signals the remote broker-thread to announce at it will send data. When the broker-thread receives the signal in the line 8 it access the container and receives the data pushing the content and signaling the local receiver tasks (lines 11 to 13).

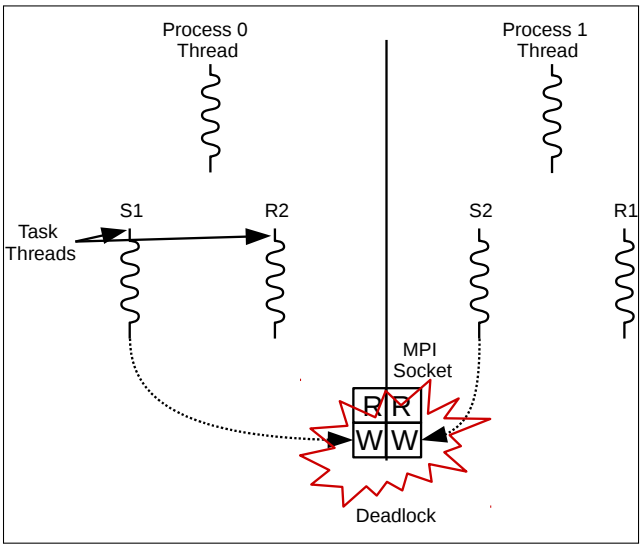


Figure 5.6: Deadlock in internode data copy. Task S1 try to reach task R1 through the interprocess MPI socket and task S2 also try to reach task R2 but no reading mode is activated on any side.

The `MCS_lock` procedure works using a shared memory space between two remote nodes and the basic idea consists in let the requester access the lock if is available or stay in a queue for the current holder. The `MCS_unlock` procedure releases the lock and signals the next requester in the queue. If no request is pending it must ensures that there no exists pending requests in transit.

5.4.2 Collective Operations

The collective operations component implements the functions requiring the participation of all the tasks in the system. Those function must be completed efficiently as possible and with minimal user interaction. The collective component implements the following collective operations: `reduce`, `allreduce`, `gather`, `scatter` and `broadcast`.

Al the collective operations implemented in XSCALA follows the MPI semantics but using tasks and trays instead of processes and buffers for example, the `reduce` depicted in the Figure: 5.7 takes as input the source tray the destination tray and the root task, to perform the addition of all the elements of the tray.

The `allreduce` performs the same operation that in `reduce` but keeping a copy of the resulting tray on each task.

The `gather` and `scatter` operations are specially designed to work with large data files, thus the first step consists in splitting the data file in smaller blocks. This partition must keep the coherence of the data being divided regardless of the number and location of

```
mutex synchMutex, DMutex /*DMutex is a distributed mutex*/
condition condN
semaphore FULL, DRecvRequest /*DRecvRequest is a distributed semaphore*/
```

```
      /*On sender task*/
 1: lock(synchMutex) /*One local task try to send at a time*/
 2: MCS_lock(DMutex) /*sender try gain the distributed mutex*/
 3: Signal(DRecvRequest) /*signals on the remote semaphore*/
 4: Send()
 5: MCS_unlock(DMutex)
 6: unlock(synchMutex)
     /*On broker-thread*/
 7: while TRUE do
 8:    wait(DRecvRequest) /*Wait for remote request*/
 9:    lock(synchMutex) /*Broker gains access to add content to the container*/
10:    Receive()
11:    pushNewContainer(TAG=receiverID, Data)
12:    cond_broadcast(condN,synchMutex)
13:    signal(FULL)
14:    unlock(synchMutex)
15: end while
     /*On Receiver task do the same as in interdevice*/
```

```
Procedure MCS_lock(DMutex)
 1: Get the LRR from DMutex Home /*LRR= Last Rank that Requested the mutex*/
 2: if LLR != -1 then
 3:    Enqueue my request in LRR /*The mutex is locked*/
 4:    Sleep
 5: else
 6:    return /*I got the mutex*/
 7: end if
```

```
Procedure MCS_unlock(DMutex)
 1: Dequeue pending requests
 2: if No pending requests then
 3:    Get the LRR from DMutex Home /*Lets ensure the are not request in transit*/
 4:    if LRR != myRank then
 5:       while TRUE do
 6:          Dequeue pending requests
 7:          if pending requests then
 8:             Break;
 9:          end if
10:          Continue /*Check requests again*/
11:       end while
12:       Signal pending request
13:    else
14:       return /*There are no pending requests*/
15:    end if
16: else
17:    Signal pending request
18: end if
```

Algorithm 5: XSCALA's algorithm for inter node data copy

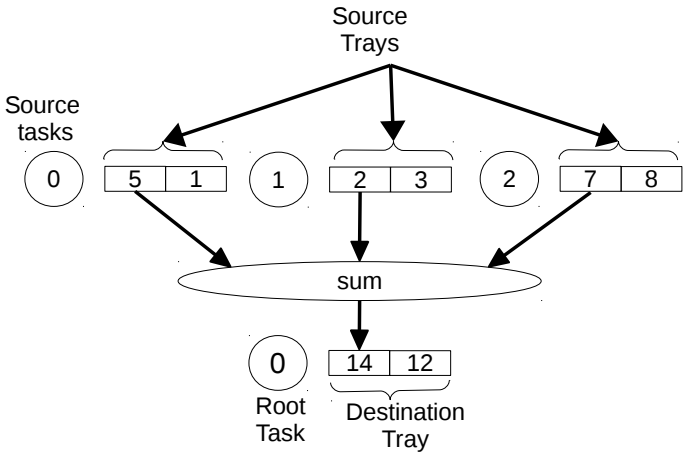


Figure 5.7: Task reduction collective operation in XSCALA.

the tasks, therefore the user is encouraged to first define the appropriated "*Entity*", the indivisible block of data defined in section 4.2.1, enabling this component to perform the possible noncontinuous file reads required for an accurate distribution.

In the `scatter` operation blocks of entities are transferred to the memory tray specified by the user. In the `gather` operation the inverse process is performed, each task first sends a block of entities to the root that then merges all this data into a single file.

Finally the `broadcast` operation copy the contents of the selected tray in the root task into the tray of all other tasks.

5.4.3 Tray Management

From the user perspective each task have a collection of trays attached to it, however multiple task could be in the same device. To keep the data of each task isolated the tray manager arranges the data in a special structure called "*rack*" as is depicted in Figure: 5.8. The rack structure keeps the data of each task organized even if multiple tasks are sharing the same device.

The tray management component is thus the responsible for allocate and free the trays on the device hosting the task. Additionally this component implements a garbage collector mechanism. The objective of the garbage collector is to keep track of the state of each task and of the memory resources no longer used to release them.

The garbage collector gets a copy of the adjacency matrix stored in the scheduler module which is updated with the operations performed by the data transfer component as follows: Every time that a send receive operation is completed the tray manager deletes

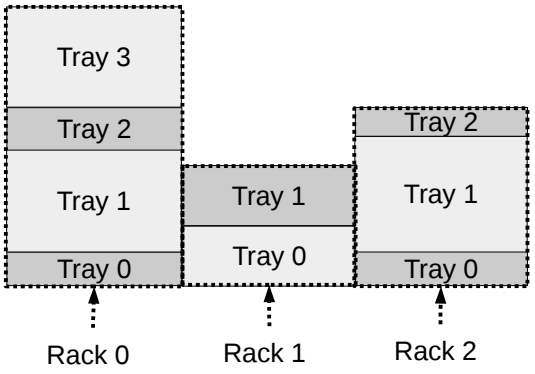


Figure 5.8: Organization of the data used by multiple task within a single device from the runtime system perspective.

the data dependency from the dependency matrix, if no additional dependencies are found the tray is automatically deleted from the device and the space is recovered.

This mechanism helps to improve the use of the memory space, a scarce resource that must be used as efficiently as possible.

5.5 The Scheduling Module

This module have two major objectives: It is responsible for constructing the mapping of tasks over computing devices and for keeping track of the location of each task in the system.

The scheduling module implements the mechanism to enable the selection of a scheduling strategy: manual, static, or dynamic. Each strategy is implemented in a different component. The fourth component is the benchmark and profiling component.

The manual component is responsible for parsing the configuration file to build the task map, the static and dynamic components contains the implementation of specific scheduling algorithms, and the benchmark and profiling component is responsible for executing the benchmarks and storing profile information. This information useful to improve the scheduling algorithms and to show the behavior of the application to the user.

The architecture of the scheduling module is depicted in the Figure: 5.9 and the details of its components is presented in the following sections.

To keep track of the location of each task the scheduling module registers each task with a unique ID in a distributed directory. The distributed directory is implemented using a indirect address table.

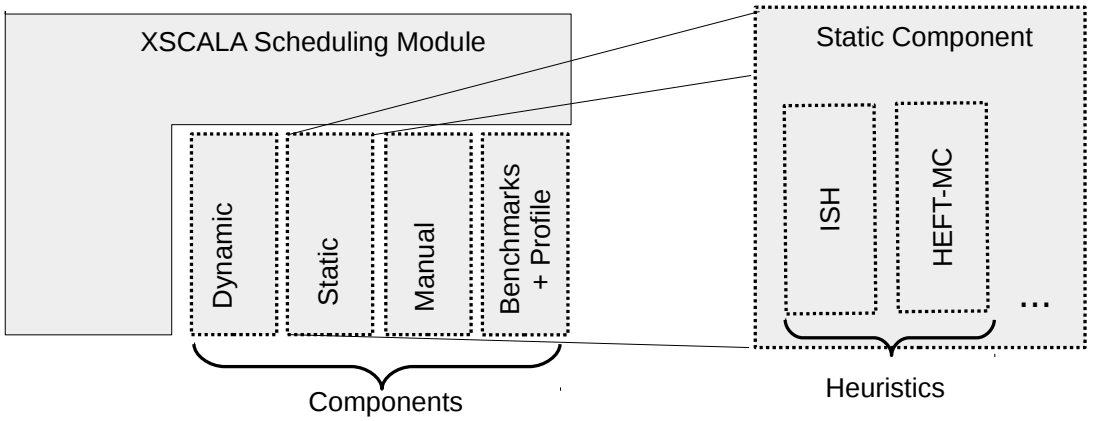


Figure 5.9: Architecture of the XSCALA scheduling module.

An indirect address table is composed of two tables having a column in common, the index table and the local table. The index table is depicted in the left hand side of the Figure: 5.10 and contains the global ID of each task, the ID of the node where the task is allocated and a local ID assigned to the tasks. This table is shared between all the nodes and must be kept in sync. The local table is depicted in the right hand side of the same figure and contains more details about the task for example the device, the memory rack, the number of trays currently used by each task, etc. This table can be updated without requiring coordination with other nodes improving the performance of the directory.

Global Task ID	Rank	Local Task ID
0	0	0
1	0	1
2	1	0
3	1	1
4	1	2
5	2	0
6	2	1

Local Task ID	Device ID	Rack Assigned	# Of Trays
0	CPU 0	0	2
1	GPU 0	0	3
2	GPU 1	1	2

Figure 5.10: Indirect address table used to implement the distributed directory. The table in the left hand stores the rank and the local ID of the task, and the table in right hand stores more details about the task like the device and memory rack assigned to the task.

To create the index table the scheduler first asks the number and type of devices available in the node to the device manager library (see section 4.1.2, on page 45) and then executes the appropriated scheduling algorithm. Finally the scheduling component uses the MPI services to synchronize the information of the index table with all the other nodes. The local table is created on each node based on the number of devices and the mapping decisions taken by the scheduler and is used to keep the memory trays aligned optimizing the use of the memory on the device.

5.5.1 Manual Scheduling

This component can be used when the user have certain knowledge about the number of tasks to be executed by the application and knows the features of the computing system for example, the number of computing devices and their features and can represent the best alternative to optimize the performance of applications.

This component also enables the design of applications were multiple requests of executions of the same task can be performed varying only the data or the procedure between each call.

To use manual scheduling the user must provide a configuration file. The syntax of each line in the configuration file is defined by the strings generated with the Context Free Grammar (CFG) of the Figure: 5.11.

```
<S> --> <Task> <Rank> <DeviceType> <DeviceID>
<Task> --> <STask> | <BTask>
<STask> --> <ID> | <ID> , <STask>
<BTask> --> <ID> - <ID>
<Rank> --> <Digit><Digit*>
<DeviceType> --> CPU | GPU | ACCEL
<DeviceID> --> <Digit><Digit*>
<ID> --> <Digit><Digit*>
<Digit*> --> <Digit><Digit*> | epsilon
<Digit> --> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Figure 5.11: CFG for XSCALA configuration file

Here `<S>` is the start symbol of the grammar, `<Task>` represent the ID of the tasks to be created, `<Rank>` is the ID of the node, `<Device Type>` is the type of device, and `<Device ID>` is the ID of the device within the node where the task will be allocated. `<STask>` and `<BTask>` are nonterminal symbols used to declare single tasks and collections of tasks respectively.

The manual scheduling component reads and parses the configuration file to build the mapping of the tasks with the desired device. An example of a configuration file is depicted in the Fig. 5.12.

5.5.2 Static Scheduling Component

This component can be used when the number of tasks, and the data dependencies among them are fixed. By default the static scheduling component implements two algorithms: HEFTMC and ISHMC. The former is a static scheduling algorithm based on

```
# task configuration file
# TaskID    Rank    Device
  0          0      GPU 0
  1,3,11     0      CPU 0
  2,4        1      GPU 1
  5-10       2      GPU 0
#end config file
```

Figure 5.12: Structure of the configuration file. This file enables to map tasks to specific computing devices.

the heterogeneous earliest finish time (HEFT) algorithm proposed in [Topcuouglu et al., 2002] and whose original complexity is $O(m \cdot e)$ where m is the number of devices in the system and e is the number of edges in the dependency graph. A major restriction to schedule tasks on H/H systems not covered by the HEFT algorithm is related to the limits imposed by the amount of memory space available on each device. To overcome this limitation the HEFT has been extended in HEFTMC to deal with memory constraints using two additional stages: first it searches those devices with enough memory to allocate the *i-th* task (T_i) decreasing the required space on the matched device. On the second stage the algorithm searches those tasks without any dependency to free the memory allocated on the device. The modified version of HEFT is called the HEFTMC and its complexity is $O(m(n + e))$ with n the number of tasks (see section 6.1.2, on page 88).

The second algorithm implemented is called ISHMC and is based in the insertion scheduling heuristic but also is extended to deal with memory constraints, The complexity of this algorithms is $O(m \cdot e + n^2)$ (see section 6.1.3, on page 91).

5.5.3 Dynamic Scheduling Component

This component is used when neither the number of tasks to be executed nor the dependencies among them are known in advance therefore the scheduling decision must be taken at runtime.

By default this component implements the Round Robin under Memory Constraints (RRMC) scheduling heuristic and the Earliest Start Time under Memory Constraints (ESTMC) heuristic. RRMC first enumerates all the devices and assigns the next request to the following device with enough memory in a sequential order (see section 6.2.1, on page 93). The ESTMC heuristic assigns the task to the device where the task can start as soon as possible, regardless of the type and location of the device, when all the devices are busy it must wait for the former that becomes idle to make the mapping (see section 6.2.2, on page 96).

5.5.4 Architecture of the Scheduling Component

The internal architecture of the dynamic and static scheduling components was designed to ease the integration and testing of new scheduling strategies. The architecture is depicted in the Figure: 5.13 and is composed of three interfaces: `I_Wrapper, I_Component and I_Scheduler` and the `AbstractComponentWraper` class that implements the `I_Wrapper` interface. The *DefaultComponent* class is the default component implemented by XSCALA and can be substituted with the one provided by the user. Similarly the *ISHMC* and *HEFTMC* classes implements the scheduler interface in XSCALA but the implementation can be provided by the user using his own heuristics. In general a developer wanting to build its own scheduling component only must implement the `I_Component` and the `I_Scheduler` interfaces.

The use of this architecture is as follows: First the client *i.e.* the scheduling module, uses the `I_Wrapper` interface to call the `newScheduler` and create a new scheduler object passing as parameter the name of the heuristic to be used. This call triggers the creation of a new instance of the component specified by the user, by default XSCALA loads the `DeafultComponent` but any user component can be loaded by XSCALA calling the `componentConstructor` function in the component. Once the component is loaded a new instance of a scheduler object is created and returned to the client who can use it to call the `matchMake` function and retrieve the desired task mapping.

5.5.5 Benchmark and Profiling

This component measures the latency in the communications between pairs of components, the compute capabilities of each device, and the amount of memory on each device. Those measures are fundamental to achieve an efficient load balancing.

In the case of communications two measures are taken: The latency measured in seconds and bandwidth measured in (GB/s). Those measures are taken sending some blocks of data between pairs of devices and measuring the time required to complete the transfer. The available amount of memory is obtained querying the device driver. The compute capability is measured in *Gflops* and is estimated by executing three subroutines of the basic linear algebra system (BLAS) [Lawson et al., 1979], measuring the time required to complete each execution.

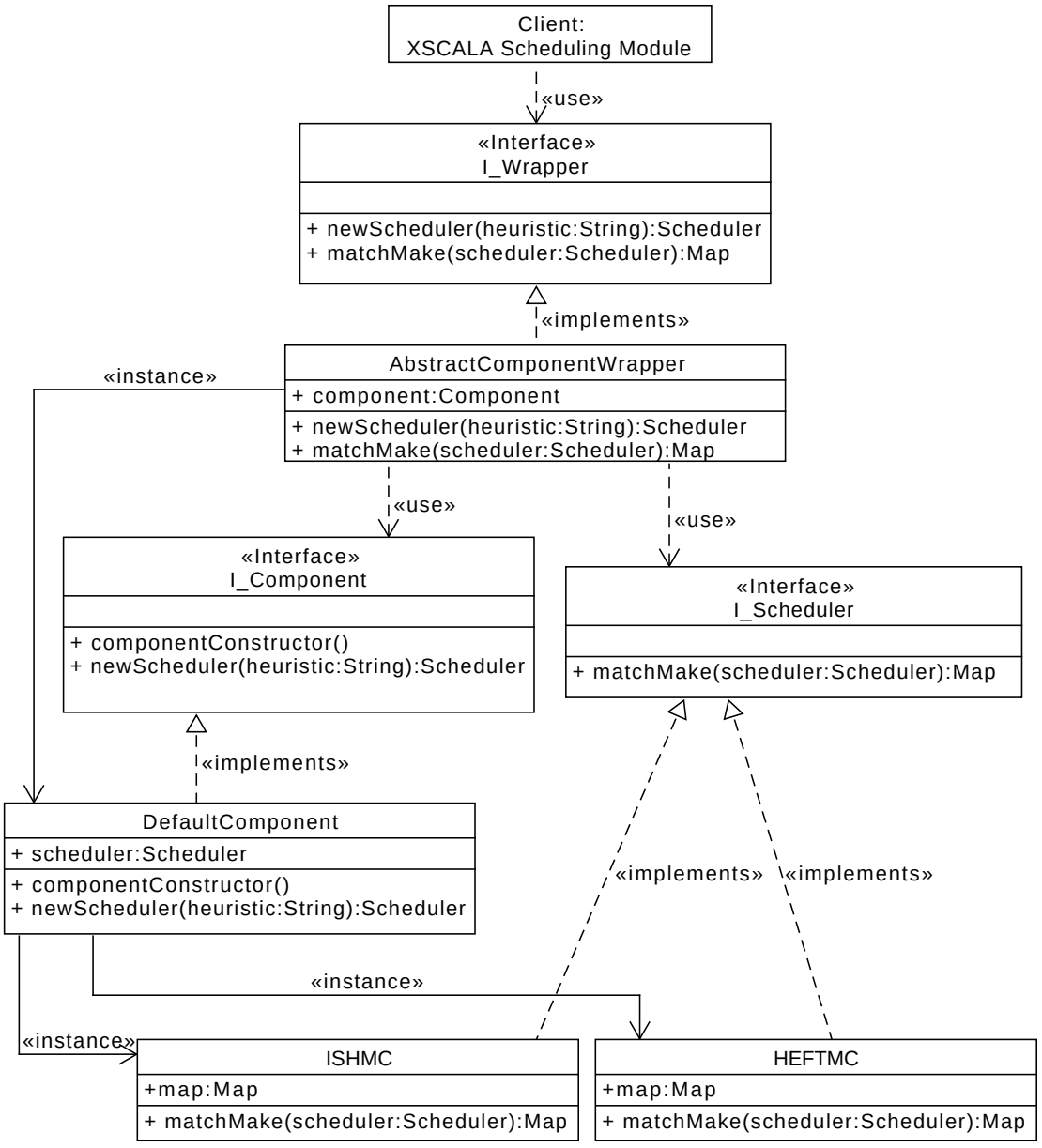


Figure 5.13: XSCALA's extensible scheduling engine.

5.6 Lifecycle of a Task in XSCALA

In order to formalize the execution model we present the lifecycle of a task in XSCALA. The lifecycle represents the set of states that each task must complete before and after the execution of its procedure.

Each task in XSCALA follows the lifecycle depicted in the Figure: 5.14 that involves the following states: registered, allocated, delegated, fetching, executing, waiting and finished.

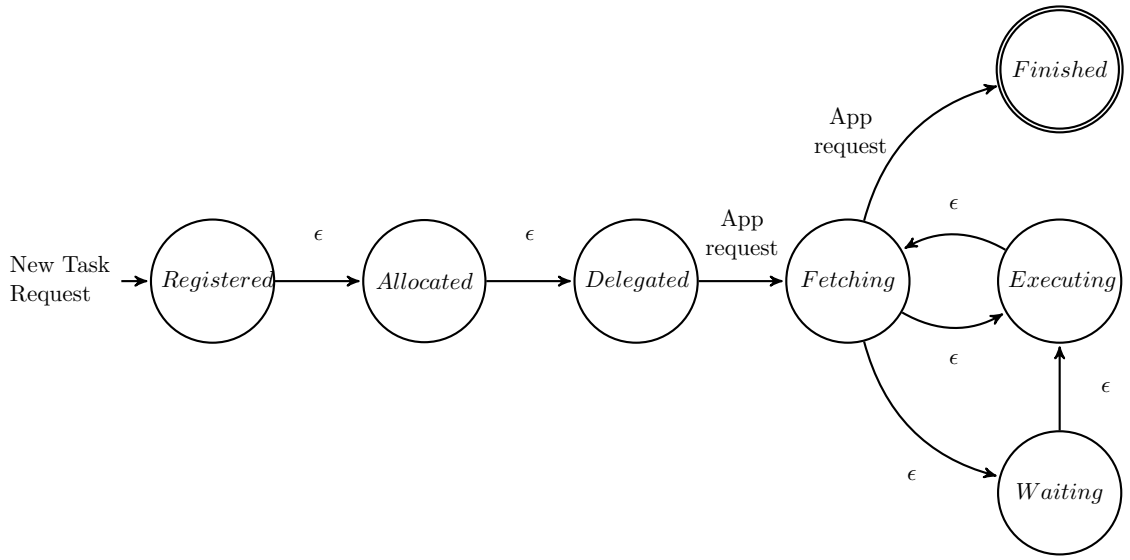


Figure 5.14: The life cycle of a tasks

Some transitions are performed after an explicit request of the application for example, the execution of the procedure or the finalization of the task, while others are performed automatically by the modules of the XSCALA middleware for example the assignation of the ID and the allocation. The description of the states and transitions between them is as follows:

1. **Registered:** Each task in XSCALA is automatically registered in the system with a unique taskID, a global identifier assigned to each task. The ID is stored in the address table described in section 5.5 and is shared between all the nodes. The register of the task is controlled by the scheduling module (see section 5.5, on page 74).

2. **Allocated:** In this state the scheduling module assigns the task to the device that fulfills its requirements. If the allocation fails for example, due to the lack of memory space, the runtime must halt and report the cause of the fail. The allocation of the task is controlled by the scheduling module (see section 5.5, on page 74).

3. **Delegated:** Once the task is registered and allocated the threads pool manager receives a request to integrate a new task-thread, when this process is completed all the subroutines related to this task will be executed by its task-thread. The task delegation is controlled by the threads pool manager (see section 5.2, on page 60).

4. **Fetching:** In this state the task checks if a new request is pending in its work queue. If is true it will try to access the resources required for its execution or

will wait if they are not available. If no more request are pending it just waits. Fetching is controlled by the threads pool manager (see section 5.2, on page 60).

5. **Executing:** In this state the task-thread is performing one of the subroutines of its command queue for example, receiving data, executing the procedure, sending data etc. The executing state is controlled by the task manager (see section 5.3, on page 64) and the data manager module (see section 5.4, on page 66).

6. **Waiting:** A task can arrive automatically to this state by several reasons: when the device is busy, when its data dependencies are not solved or if there is no more pending work. This state also is used to synchronize the execution of the tasks. The waiting state is controlled by the task manager (see section 5.3, on page 64).

7. **Finished:** In this final state the task-thread concluded the execution of all the subroutines delegated to the task. Next the task releases all its resources for example, the memory space and it is permanently removed from the system. The task can enter to this state only under a explicit request from the application. The Finished state is controlled by the task manager (see section 5.3, on page 64) and the threads pool manager (see section 5.2, on page 60).

5.7 Correctness of the Execution

A major objective of a parallel task programming model is to let the programmer assume that certain events just happen and he do not be worried about how they actually happen for example, we must let the programmer to assume that each task will be eventually executed in a processing unit and that the data transfers will be eventually completed no matter how, however, we must find out the mechanisms to ensure that all the events represented by the model will be executed in the correct order and as efficiently as possible.

The services supported by the framework are designed to ensure compliance of data dependencies and to guarantee correctness in the execution. In particular we need to ensure that the following statements are met:

1. Each task must be eventually executed $\begin{cases} \text{No starvation.} \\ \text{No deadlock.} \end{cases}$

2. In the correct order $\begin{cases} \text{Meeting precedence constraints.} \\ \text{No data races.} \end{cases}$

3. As efficiently as possible
   - Trying to minimize makespan.
   - Choose the faster data copy mechanism.
   - Overlap computation with communication.

Lets now present formal definitions for each of these requirements followed by the algorithms implemented in XSCALA to ensure its compliance.

The execution of an application in a concurrent system can be modeled by a history, a finite sequence of method invocation and response events [Herlihy and Shavit, 2012]. The formal description of the condition "*Each task must be eventually executed* " is as follows: Let $\Pi = \{T_1, T_2, \ldots, T_m\}$ to represent the set of all task, τ to be a logical clock and let $\mathcal{C}_t : t \to 2^{\Pi}$ to be a function denoting the subset of tasks executed up to time t the following constraint must be meet:

$$\forall T \in \Pi, \exists t \in \tau \mid T \in \mathcal{C}_t \text{ and } t < \infty \tag{5.1}$$

The algorithms 3, 4 and 5 in the data management module ensure that no task stays in starvation waiting for its dependencies while the algorithm 1 in the thread pool manager is the responsible for ensure each task can eventually request its execution and access resource like the communication channel or the computing device.

The formal definition of the condition "*In the correct order*" is as follows: Let $\mathcal{E}^k$ to represent the event: execution of task k, and s^m and r^m to represent the events send of message m, and receive of message m between the tasks T_i and T_j and lets assume that T_j has a dependence on T_i therefore the following relation is always meet:

$$\tau(\mathcal{E}^i) \prec \tau(s^m) \prec \tau(r^m) \prec \tau(\mathcal{E}^j) \tag{5.2}$$

The receiving side in the algorithms 3, 4 and 5 blocks the advance of the tasks until all their dependencies are met, and the algorithm 2 in the data management module is responsible for establish synchronization points avoiding data races between the process-thread and the task-threads when the former performs a shared data access.

Finally the formal definition of the "*As efficiently as possible*" condition can be represented by the scheduling problems in particular XSCALA deals with the $R|\mathbf{res}\ 1s[\cdot]r[\cdot], \mathbf{prec}|C_{max}, WS$ scheduling problem (see section 6.1, on page 85) trying

to find out good solutions for the following optimization problem:

$$min \quad \{C_{max}, WS\} \tag{5.3}$$

s.t.

$$\sum_{T_i \in A} r[T_i] \leq s^0[j]. \tag{5.4}$$

Where $C_{max} = max\{C_j \mid j = 1, \ldots, n\}$ represents the time of finalization of each task, WS represents the memory space unused (wasted), r represents the memory requirement of each task and s represents the memory space available on each device.

Here the ISHMC, HEFTMC (see section 6.1, on page 85), RRMC and ESTMC (see section 6.2, on page 93) algorithms of the scheduling module are the responsible for distributing the workload as efficiently as possible.

In summary the algorithms implemented in the XSCALA middleware can ensure the correctness of the execution based on the requirements pointed out before.

Chapter 6

Scheduling

The performance achieved in the execution of an application is strongly related with the technique used for workload distribution. A workload distribution must avoid overloading a single computing device beyond its capabilities nor allocating tasks with intensive data communication in remote nodes connected through low bandwidth channels.

The problem of matching task to devices in an optimal way is known as the scheduling problem and due to its complexity is strongly believed that there no exists an algorithm that can find optimal mappings using reasonable amounts of time. Several algorithms based on heuristics have been developed for specific cases providing good solutions in short amounts of time.

In this chapter we analyze and propose the algorithms to solve the problem of static and dynamic scheduling under memory constraints. We first introduce the characteristics of the problem, and then we present the algorithms implemented in XSCALA to solve each problem.

6.1 Static Scheduling Algorithms

The static scheduling algorithms can be used when the number of tasks, and the data dependencies among them are fixed. The static algorithms can analyze the provided information to find a mapping that minimizes the completion time.

Our scheduling algorithms implement the phases found in other well known scheduling algorithms like task prioritizing and processor selection but they are enhanced to deal with the memory constraints using two additional phases: device filtering and memory recovery.

6.1.1 The R|res $1\cdot\cdot$,prec|C_{max}, WS Scheduling Problem

A common approach to solve the static scheduling problem consists of using greedy algorithms like the Heterogeneous Earliest Finish Time (HEFT) [Topcuouglu et al., 2002] and the Insertion Scheduling Heuristsc (ISH) [Kruatrachue and Lewis, 1987]. The basic idea behind these heuristics is as follows: Assign priority to the tasks based on their distances to the exit point and schedule tasks with higher priority first.

The major restriction to implement the schedules based on these algorithms is that they are designed for the case of unrelated parallel machines with precedence constraints(**R**|**prec**|C_{max}) and homogeneous parallel machines with precedence constraints (**P**|**prec**|C_{max}) problem respectively and they do not consider the limitation in the amount of memory available on each device, hence, the schedules that result from these algorithms might be unsuitable at runtime if they overload the faster device with a number of tasks that exceeds its memory capabilities.

To handle this limitation we add the restriction **res** $1\cdot\cdot$ to the scheduling problem. The new scheduling problem consists of unrelated parallel machines subject to memory and precedence constraints and our proposal to solve it is as follows:

Let $G_t(V_t, E_t, c_t, w)$ to represent the directed acyclic graph of an application and $G_r(V_r, E_r, c_r)$ to represent the resources graph with V_t,E_t, c_t, w, V_r, E_r and c_r as they were defined in section 2.5.2, $m = |V_r|$ to represent the number of devices in the system, $n = |V_t|$ the number of tasks to be scheduled and $r[\cdot]$ to be a vector representing the amount of memory required by each task in G_t.

Considering that the strategy of a greedy algorithm consists in mapping a task i with a computing device j on each step of the process lets use k as an integer counting the number of steps executed and let $s^k[\cdot]$ to be a vector representing the amount of memory available on each device of V_r in the step k of the scheduling process where $k \geq 1$ defining s^0 as follows:

$$s^0[\cdot] = \text{Total amount of memory in each device.} \tag{6.1}$$

In the *device filtering phase* we select those devices with enough memory to allocate the next task to be scheduled *i.e.* for any given a task i that needs to be scheduled we build a subset Q of V_r as follows:

$$Q = \{j \in V_r \mid s^k[j] \geq r[T_i]\}; \quad \forall k > 0. \tag{6.2}$$

Where: $s^k[j]$ represents the amount of memory space available in the device j in the step k of the scheduling process, and $r[T_i]$ represents the amount of memory required by the task i. If a task i is mapped with the device j we update the vector s subtracting the amount of memory occupied by the task as follows:

$$s^{k+1}[j] = s^k[j] - r[T_i]; \tag{6.3}$$

Given that the memory is a scarce resource we must try to maximize its use even at the cost of loosing some performance in the makespan *i.e.* we must find a trade off between two conflicting objectives: minimization of the total execution time and maximization of memory occupancy.

We can model our new optimization criteria as follows: Let A_j^k to represent the set of tasks mapped to the j^{th} device in the k^{th} scheduling step. We define the occupancy of the device j at step k as follows:

$$O_j^k = \sum_{T_i \in A} r[T_i]; \tag{6.4}$$

Given that the total amount of memory on each device might be different we define the ratio of occupancy of the device j in the step k as:

$$R_j^k = \frac{O_j^k}{s^0[j]}. \tag{6.5}$$

In order to establish the corresponding minimization problem we define the ratio of wasted memory (WS_j^k) as follows:

$$WS_j^k = 1 - R_j^k. \tag{6.6}$$

The new optimization criteria consists in minimize WS is defined as follows:

$$WS = \sum_{T_i \in V_t} WS_{X(T_i)}^k \quad k = 1, \ldots, n \tag{6.7}$$

Where $WS_{X(i)}^i$ represents the ratio of memory wasted in the mapping of the task i into the device $X(T_i)$. The single optimization criteria C_{max} is thus replaced with the following two objective optimization criteria.

$$min \quad \{C_{max}, WS\} \tag{6.8}$$

$$\text{s.t.}$$

$$\sum_{T_i \in A} r[T_i] \leq s^0[j]. \tag{6.9}$$

Where $C_{max} = max\{C_j \mid j = 1, \ldots, n\}$ and $WS_i = 1 - R_i$. This new criteria can be expressed as:

$$R|\mathbf{res}\ 1s[\cdot]r[\cdot], \mathbf{prec}|C_{max}, WS \tag{6.10}$$

Finally in the memory recovery phase we perform a backward search among the tasks already scheduled without pending dependencies to recover the memory space previously used. In the following sections we present two algorithms to solve this scheduling problem.

6.1.2 The HEFTMC Algorithm

We now present the Heterogeneous Earliest Finish Time subject to Memory Constraints (HEFTMC) scheduling algorithm that solves the static scheduling problem for unrelated parallel machines having memory constraints. Our algorithm is based on the HEFT algorithm that solves the **R**| **prec** $|C_{max}$ scheduling problem but extended to deal with the memory constraints.

Lets first introduce some attributes of the task graph. The average computation cost of the i^{th} task ($\overline{w_i}$) is defined as:

$$\overline{w_i} = \sum_{j=1}^{m} \frac{w_{i,j}}{m}. \tag{6.11}$$

With m and $w_{i,j}$ as defined before. The communication cost c of the edge (a, b), for transferring data from task a (scheduled on u) to task b (scheduled on v), is defined as:

$$c_{a,b} = L_{u,v} + \frac{data_{a,b}}{B_{u,v}}. \tag{6.12}$$

Where $L_{u,v}$ represents the latency and $B_{u,v}$ represents the bandwidth between the processing units u and v respectively and $data_{a,b}$ represents the amount of data to be transferred measured in *Bytes*.

The average communication cost of an edge (i, k) defined as:

$$\overline{c_{a,b}} = \overline{L} + \frac{data_{a,b}}{\overline{B}}. \tag{6.13}$$

Where $\overline{L}$ represents the average latency measured in *seconds* (s) and $\overline{B}$ represents the average bandwidth measured in bytes per second (B/s).

HEFTMC solves the $\mathbf{R}|\mathbf{res}\ 1\cdot\cdot,\mathbf{prec}|C_{max}, WS$ problem implementing the four phases pointed out before: task prioritizing, device filtering, processor selection and memory recovery.

(i) **Task prioritizing.** Let the upper rank of a task i denoted as $rank_u(T_i)$ to represent the priority of the i^{th} task. The upper rank measures the worst case of the time required to reach the end of the exit task from the task i and is calculated starting from the exit task using the following recursive relation:

$$rank_u(T_i) = \overline{w_i} + \max_{T_j \in Succ(T_i)} \{\overline{c_{ij}} + rank_u(T_j)\}. \tag{6.14}$$

(ii) **Device filtering.** Here we use the current values in the vector s to build the subset of devices able to allocate the task using the relation 6.2.

(iii) **Processor selection.** This phase requires to compute two attributes for each task: the earliest start time (EST) and earliest finish time (EFT) representing the minimum time required to initialize and to finalize the execution of a tasks in a processing unit. These attributes are computed as follows:

$$EFT(i, j) = w_{i,j} + EST(i, j). \tag{6.15}$$

$$EST(i, j) = max\{avail[j], \max_{T_m \in Pred(T_i)} \{AFT(T_m) + c_{m,i}\}\}. \tag{6.16}$$

Where $avail[j]$ is the time at which the device j completed the last task assigned to it, and $AFT(T_m)$ measures the time required to complete the task m preceding T_i.

As we pointed out before we must search a trade off between the minimization of the execution time and the minimization of memory wasted. To solve this optimization problem we first define the following attributes:

$$O_{rel}(i,j) = \frac{r(T_i)}{s^0[j]} \tag{6.17}$$

$$W_{rel}(i,j) = R_j^k - O_{rel}(i,j) \tag{6.18}$$

$O_{rel}(i,j)$ represents the relative occupation of the task i with respect to the space available in the device j and $W_{rel}(i,j)$ represents the relative waste of space reached if the task i is allocated in device j.

Now we build a weighted sum of the objectives to define a single optimization problem. Let f to be a function defined as follows:

$$f(g_1, g_2, \lambda) = \lambda g_1 + (1-\lambda) g_2 \tag{6.19}$$

Where:

$$g_1 = \frac{EFT(i,j)}{EFT^*(i)}, \tag{6.20}$$

$$g_2 = \frac{W_{rel}(i,j)}{W_{rel}^*}, \tag{6.21}$$

$$0 \leq \lambda \leq 1. \tag{6.22}$$

Here EFT^* represents the best finish time that can be reached for the task i, and W_{rel}^* represents the minimal waste of memory that can be reached at the time of scheduling the task i. Our processor selection criteria consists in mapping the task i with the device j that minimizes f.

(iv) **Memory recovery.** Our algorithm searches if there exists a tasks with no pending memory requests to recover the memory space. This can be done by looking backward for the tasks already scheduled and without data dependencies *i.e.* where the current out degree of the tasks equal to zero.

The HEFTMC is shown in the Algorithm: 6 and work as follows: In lines 1 to 3 we perform the task prioritizing phase, line 2 sets the upper rank and out degree of each task, this operation can be done in $O(|V_t| + |E_t|)$ and sorting in line 3 can be done in $O(log(n))$. Line 6 performs the device filtering phase which is performed in $O(m)$. Lines 8 to 11 perform the processor selection, this operation takes $O(m \cdot I_{Deg}(T_i))$ where $I_{Deg}(T_i)$ represents the in degree of the task i. In line 12 we subtract the amount of memory required by the task. Lines 13 to 18 perform memory recovery, in this process the predecessors of the scheduled task are evaluated to known if they do not

have more dependencies and recover its memory space. This operation can be completed in $O(I_{Deg}(T_i))$.

Input: $w(i,j)$ = Execution cost matrix
G_t = Task Dependency Graph
$r[\cdot]$ = Memory Space Requirement of tasks
$s[\cdot]$ = Available memory space on each device

Output: X(T_i) //Task Device Map.

1: Set the computation cost of task and communication cost the edges in G_t with mean values.
2: Compute the upper rank ($rank_u$) and out-degree ($O_{Deg}(v)$) for each vertex in G_t by traversing it upward starting from the exit task.
3: Sort the tasks in a scheduling list L_T by non-increasing order of $rank_u$ values.
4: **while** Exists unscheduled tasks **do**
5: Select the first task T_i from L_T.
6: Let $Q = \{j \in V_r \mid s[j] \geq r[T_i]\ \}$ /*Device filtering*/
7: **if** $Q \neq \{\emptyset\}$ **then**
8: **for each** j **in** Q **do**
9: Compute f.
10: **end for**
11: Map T_i to the $j \in Q$ that minimizes f
12: $s[X(T_i)] \leftarrow s[X(T_i)] - r[T_i]$
13: **for each** T_m **in** predecessors of T_i **do**
14: $O_{Deg}(T_m) \leftarrow O_{Deg}(T_m) - 1$
15: **if** $O_{Deg}(T_m) == 0$ **then**
16: $s[X(T_m)] \leftarrow s[X(T_m)] + r[T_m]$ /*Memory recovery*/
17: **end if**
18: **end for**
19: **else**
20: **Return** Error: Not enough space to schedule all tasks.
21: **end if**
22: **end while**
23: **Return** X /*Task-Device Matchmaking list.*/

Algorithm 6: Heterogeneous Earliest Finish Time Algorithm Under Memory Constraints (HEFTMC)

The complexity of the processor selection phase in the HEFT algorithm is $O(m \cdot e)$ with e the number of edges in the G_t however the HEFTMC algorithm has a complexity of $O(m(n+e))$ for this phase. This increment although small is explained by the integration of the device filtering and memory recovery phase.

6.1.3 The ISHMC Algorithm

We now present the Insertion Scheduling Heuristic subject to Memory Constraints (ISHMC) scheduling algorithm that solves the static scheduling problem for homogeneous parallel machines having memory constraints ($\mathbf{P}|\mathbf{res}\ 1\cdot\cdot,\mathbf{prec}|C_{max}, WS$).

This algorithm is based in the ISH algorithm that is designed to solve the $\mathbf{P}|\mathbf{prec}\ |C_{max}$ scheduling problem but extended to deal with memory constraints. Similar to the case of the HEFTMC, the ISHMC algorithm filters the devices with enough memory space and try to recover memory space looking for tasks without data dependencies.

(i) **Task prioritizing.** We first assign the $b-level$ of each task. As the upper rank the $b-level$ attribute is computed recursively from the exit task using the following recursion:

$$b-level(T_i) = w_i + \max_{T_j \in Succ(T_i)} \{c_{ij} + b-level(T_j)\}. \tag{6.23}$$

Unlike the upper rank attribute the b-level assumes that the processing units and the communication channels have similar capabilities.

(ii) **Device filtering.** Here we also use the updated vector s to build the subset of devices able to allocate the task using the equation 6.2.

(iii) **Processor selection.** This phase requires to compute the earliest start time (EST) of each task. This attribute is computed as follows:

$$EST(i,j) = max\{avail[j], \max_{T_m \in Pred(T_i)} \{AFT(T_m) + c_{m,i}\}\}. \tag{6.24}$$

With AFT the finish time of each of the predecessors of T_i. The optimization function for this algorithm is defined as follows:

$$f(g_1, g_2, \lambda) = \lambda g_1 + (1-\lambda) g_2 \tag{6.25}$$

Where:

$$g_1 = \frac{EST(i,j)}{EST^*(i)}, \tag{6.26}$$

$$g_2 = \frac{W_{rel}(i,j)}{W^*_{rel}}, \tag{6.27}$$

$$0 \leq \lambda \leq 1. \tag{6.28}$$

Where: EST^* represents the best value of EST and W^*_{rel} is defined as in HEFTMC.

Once the task is matched with the best device we search idle slots in the device to try to insert more tasks from the ready list. An idle slot is defined as the lapse between the $AFT(T_m)$ and the $EST(i,j)$ whit T_m the last task assigned to j. If there exists a task that fits in the slot and can not start earlier in other device we

evaluate if the device still have enough memory, if true, the task is mapped to j. The new slots generated can be used latter to insert more tasks.

(iv) **Memory recovery.** In this phase we search tasks without pending requests to recover the memory space. This phase is performed as in the HEFTMC algorithm.

The ISHMC is shown in the Algorithm: 7 and work as follows: in lines 1 and 2 we perform task prioritizing, line 1 sets the b-level and out degree of each task, this operation can be done in $O(|V_t|+|E_t|)$ and sorting in line 2 can be done in $O(log(n))$. Line 6 performs the device filtering phase which is performed in $O(m)$. Lines 8 to 10 perform the processor selection, this operation takes $O(m \cdot I_{Deg}(T_i))$. In line 12 we subtract the amount of memory required by task and in line 13 we try a memory recovery that takes $O(I_{Deg}(T_i))$. Lines 14 to 22 implement the insertion step, in the worst case this process will try to fit each tasks in V_t without any success causing an additional cost of $O(e)$. Finally the line 23 updates the ready list with the nodes that are now ready to be scheduled.

The complexity of the processor selection phase in the ISH algorithm is $O(m \cdot e + n^2)$ with e the number of edges in the G_t, and in the HEFTMC algorithm processor selection has a complexity of $O(m(n + e) + n \cdot e)$ for this phase explained by the integration of the device filtering and memory recovery phase.

6.2 Dynamic Scheduling Algorithms

The use of dynamic scheduling techniques enables to solve the problem of mapping tasks when neither the number of tasks nor their dependencies are known beforehand therefore the scheduling decision must be taken at runtime. This strategy is required in applications like numerical methods and mesh refinement were more task can be required at runtime to reach certain precision.

In the following section we present two dynamic scheduling algorithms subject to memory constraints. The first is based on the Round Robin algorithm and the second is based in the Earliest Start Time (EST) algorithm.

6.2.1 RRMC Algorithm

Our implementation of the Round Robin scheduling algorithm under Memory Constraints (RRMC) consists in create a queue that includes all the devices available and assigns each task to the next device in the queue with enough memory to allocate the task regardless of their location or workload. This algorithm results suitable only for

Input: $w(i,j)$ = Execution cost matrix,
G_t /*Task Dependency Graph*/
$r[\cdot]$ //Memory Space Requirement of tasks
$s[\cdot]$ //Available memory space on each device

Output: X(T_i) //Task Device Map.

Compute the *b-level* and the out-degree $O_{Deg}(v)$ for each vertex $\in G_t$ by traversing it upward starting from the exit task
Sort the tasks by non-increasing order of *b-level*.
Set $ReadyList = \{entryTask\}$
while $ReadyList \neq \{\emptyset\}$ **do**
  $T_n := \text{pop}(ReadyList)$.
  Let $Q = \{j \in V_r \mid s[j] \geq r[T_i]\ \}$ /*Device filtering*/
  if $Q \neq \{\emptyset\}$ **then**
    for each j **in** Q **do**
      Compute f
    end for
    Map T_i to the device $\in Q$ that minimizes f
    $s[X(T_i)] \leftarrow s[X(T_i)] - r[T_i]$ /* *Decrease available space in device* $X(T_i)$*/
    ReclaimMemSpace(T_i)
    if Scheduling of T_i generates an idle slot in $X(T_i)$ **then**
      for each T_m **in** *ReadyList* **do**
        if (T_m fits in any slot of $X(T_i)$, **and**
          cannot start earlier in any other device, **and**
          $s[X(T_i)] \geq r[T_m]$)
        then
          Assign T_m to $X(T_i)$
          $s[X(T_m)] \leftarrow s[X(T_m)] - r[T_m]$
          ReclaimMemSpace(T_m)
        end if
      end for
    end if
    Update the *ReadyList*
  else
    Return Error: Lack of space to schedule all tasks.
  end if
end while
Return X.

Procedure ReclaimMemSpace(T)
for each T_m **in** predecessors of T **do**
  $O_{Deg}(T_m) \leftarrow O_{Deg}(T_m) - 1$
  if $O_{Deg}(T_m) == 0$ **then**
    $s[X(T_m)] \leftarrow s[X(T_m)] + r[T_m]$ /*Memory recovery*/
  end if
end for

Algorithm 7: The insertion scheduling algorithm under memory constraints (ISHMC)

applications with few dependencies and resources with similar capabilities resulting in a fast mapping of tasks with good distribution.

In spite of the lack of information about the dependencies in applications using dynamic

scheduling certain strategies can be used to improve the performance of the applications. The strategy followed in the RRMC algorithm consists in delaying mapping decisions until the very last moment of the execution storing the pending data receives using the *"callsBag"* mechanism of XSCALA (see section 5.2, on page 60). The information stored in the *"callsBag"* is used to compute the amount of memory required by the task.

Let r to represent the memory requirement of the task to be mapped , L_R a list of pending receives from the predecessors of the task and r_i to represent the size of each pending receive in L_R. The parameter r is computed as follows:

$$r = \sum_{r_i \in L_R} r_i \tag{6.29}$$

The Algorithm: 8 implements the RRMC scheduling algorithm and works as follows: In the line 7 we get the next device in the sequence, in line 8 we query if the device have enough memory if true, we map the task and put device in the end of the queue, and if not we iterate on the queue until we find a suitable device.

Input: m = the number of devices in the system.
$s[\cdot]$ = Available memory space on each device

Output: X(T) //Task Device Map.

```
1: Create a queue with the IDs of the devices
2: Initialize an event handler waiting for task scheduling requests
3: When the request arrives the following actions are performed
4: Set k← 0
5: Compute r
6: while (k < m) do
7:    j=queue.pop
8:    if (s[j] ≥ r) then
9:       X(T)=j /*Map T to j*/
10:      s[j] ← s[j)] − r
11:      queue.push(j) /*Send this device to the end of the queue*/
12:      Return X
13:   else
14:      queue.push(j)
15:      k←k+1
16:   end if
17: end while
18: Return Error: Not enough space to schedule all tasks.
```

Algorithm 8: Round Robin Scheduling Algorithm Under Memory Constraints

6.2.2 ESTMC Algorithm

The Earliest Start Time under Memory Contsraints (ESTMC) algorithm maps each task with the device that provides the shorted time required to start the execution. Our algorithm also delays the mapping decisions until the execution of the procedure arrives, and to get the earliest start time of the task the algorithm must find the device that provides the smaller data transfer cost. The earliest start time of a task can be computed as follows:

$$EST(j) = \max_{T_m \in Pred(T)} \{c_{m,j}\}. \tag{6.30}$$

The memory requirement r is also computed using the equation 6.29, and the implementation is depicted in the Algorithm: 9 that works as follows: In the line 3 we cmpute the memory requirement of the task to be scheduled based on the pending receives accumulated in the "*callsBag*". In line 4 we perform a device filtering and in line 5 to 7 we compute the earliest start time of the task on all the devices with enough memory. Finally line 8 assigns the task to the best device and line 9 subtract the memory used by the task form the memory of the device.

Input: V = List of available devices
$s[\cdot]$ = Available memory space on each device

Output: X(T) //Task Device Map.

1: Initialize an event handler waiting for task scheduling requests
2: When the request arrives the following actions are performed
3: Compute r
4: Let $Q = \{j \in V \mid s[j] \geq r[T]\ \}$ /*Device filtering*/
5: **for each** j **in** Q **do**
6: Compute EST
7: **end for**
8: Map T to the device $\in Q$ that minimizes EST
9: $s[X(T)] \leftarrow s[X(T)] - r$
10: **Return** X

Algorithm 9: Dynamic Earliest Start Time Algorithm Under Memory Constraints

Part III

Results

Chapter 7

Experimentation and Results

In this chapter we present several applications implemented with the XSCALA framework. The objective of this set of applications is to show the advantages of our proposal to ease the development and to demonstrate the adaptability and scalability of the applications. To evaluate the level of scalabilty we measured the performance of the applications varying the size of the problem and the number of devices in the execution environment.

The selected applications are: The Strassen algorithm for matrix multiplication, the SAXPY problem, the SGEMM problem, and the N-Body simulation problem. The applications were selected to be representative of the multiple levels of granularity varying from fine grained like in SAXPY, medium like in N-Body and coarse grained like in the Strassen algorithm or in SGEMM.

The applications are presented as follows: First we give a brief introduction about the application describing the algorithm to be implemented. Next we present the development of the application followed by the experimentation and the results, here we describe how we performed the experiments and present the graphs and tables summarizing the results. Finally we present the discussion where we analyze the behavior of the application and explain the most relevant observations.

The development of the applications in XSCALA can be completed using the following steps:

1. **Entities Definition:** Here we must define the structure of the entities to be used.
2. **Tasks Definition:** Here we define the number of tasks to be executed and the procedures assigned to each task.

3. **Data Dependencies:** Here we establish the data dependencies between the tasks.
4. **Scheduling:** Here we define the scheduling technique to be used or the workload distribution in case of using manual scheduling.
5. **Implementation:** Here we show part of the code required for the implementation of the applications emphasizing the steps performed with XSCALA function calls.

7.1 Experimental Platforms

For the execution of the applications we used different platforms representative of the multiple environments supported by XSCALA, varying from single nodes to clusters of nodes with multiple accelerators. The specific features of each platform are the following:

- **Platform A: Single node with a CPU and a GPU**

  This platform consists of a single node having one multicore processor and one GPU. The properties of each device are described in the Table: 7.1.

Component	Features
CPU	1 x Intel Core i7-3610QM @ 2.30GHz
GPU	1 x NVIDIA GeForce 640M LE
RAM	6144 MB
GPU memory	1024 MB
Network	N/A
MPI (Ver.)	Open MPI (1.8.2)
OpenCL (Ver.)	1.1 for NVIDIA GPUs and 1.2 for Intel CPU
Operating System	Ubuntu 14.04 (kernel 3.14.4)

Table 7.1: Architecture of the platform A.

- **Platform B: Single node with multiple GPUs**

  This platform consists of a single node with two multicore processor and three GPUs with heterogeneous capabilities. The properties of each device are described in the Table: 7.2.

- **Platform C: Multiple nodes with multiple GPUs**

  This platform consists of four nodes connected by a local network. The node 0 have two devices labeled as "`CPU_0`" and "`GPU_0`", The node 1 have the devices "`CPU_1`" and "`GPU_1`", the node 2 have the devices "`CPU_2`", and "`GPU_2`" and the node 3 have the devices "`CPU_3`", and "`GPU_3`". The features of each device are summarized in the Table: 7.3.

Component	**Features**
CPU	2 x Intel Xeon E5-2650 @ 2.80GHz
RAM	16384 MB
GPU 1	1 x NVIDIA GTX 460
GPU 1 memory	1024 MB
GPU 1 Architecture	Fermi (7 SMs, 336 CUDA cores)
GPU 2	1 x NVIDIA Quadro K2000
GPU 2 memory	2048 MB
GPU 2 Architecture	Kepler (2 SMX, 384 CUDA cores)
GPU 3	1 x NVIDIA Tesla C2070
GPU 3 memory	6144 MB
GPU 3 Architecture	Fermi (14 SMs, 448 CUDA cores)
Network	N/A
MPI (Ver.)	Open MPI (1.8.2)
OpenCL (Ver.)	1.1 for NVIDIA GPUs and 1.2 for Intel CPU
Operating System	CentOS 5 (kernel 3.14.4)

Table 7.2: Architecture of the platform B.

	Node 0	**Node 1,2**	**Node 3**
CPUs	1 x Intel Core i7 950 @ 3.06GHz	1 x Intel Core i7 950 @ 3.06GHz	1 x Intel Core i7 920 @ 2.67GHz
GPUs	1 x NVIDIA GeForce 8400 GS	1 x NVIDIA GeForce 8400 GS	1 x NVIDIA GeForce 8400 GS
RAM	8192 MB	4096 MB	4096 MB
GPU memory	512 MB	512 MB	512 MB
Network	ETHERNET CISCO SG300-10P		
MPI (Ver.)	Open MPI (1.8.2)		
OpenCL (Ver.)	1.1 for NVIDIA GPUs and 1.2 for Intel CPUs		
Operating System	Fedora 19 (kernel 3.14.4)	Centos 6.7 (kernel 2.6.32)	Ubuntu 13.10 (kernel 3.11.4)

Table 7.3: Architecture of the platform C.

7.2 The Strassen Algorithm

Introduction

The Strassen algorithm [Strassen, 1969] is an efficient algorithm to compute the product of two matrices $C = A \times B$ of size $N \times N$ using $O(N^{log_2 7})$ arithmetic operations. The major advantage of the algorithm is the reduction on the number of arithmetic operations but we can exploit another important advantage of the algorithm: its ability to split the workload in several independent operations of size $(N/2)$ that can be executed in parallel. The algorithm works as follows:

1. Split each matrix in four blocks as follows:

$$\begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \tag{7.1}$$

2. Perform the following block matrix operations:

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1} \\ \mathbf{M}_3 &:= \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_4 &:= \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_5 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \\ \mathbf{M}_6 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \end{aligned} \tag{7.2}$$

3. Compute the blocks of C as follows:

$$\begin{aligned} \mathbf{C}_{1,1} &:= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\ \mathbf{C}_{1,2} &:= \mathbf{M}_3 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &:= \mathbf{M}_2 + \mathbf{M}_4 \\ \mathbf{C}_{2,2} &:= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{aligned} \tag{7.3}$$

One of the problems for its implementation is the complex pattern of memory access required on each matrix multiplication which might in turn become more expensive that the simpler $O(N^3)$ *school book* algorithm.

We will use task parallelism to harness the advantages of the algorithm by creating several independent tasks assigned to multiple devices executing the operations required by the algorithm in parallel with the goal of accelerate the overall completion.

To demonstrate the advantages of XSACLA in the development of applications we will compare the number of code lines required to implement the application against the typical OpenCL + MPI approach. Next we measure the performance of the application varying the size of the matrix and the technique used for scheduling.

Development

1. *Entities Definition.* For this application we define an entity as each one of the submatrices $A_{i,j}, B_{i,j}, C_{i,j}$ in the Equation: 7.1.

2. *Tasks Definition.* In this application we define a task for each one of the operations in equations 7.2 and 7.3 for example the `Task 0` will perform $(\mathbf{A}_{1,1} + \mathbf{A}_{2,2})$, the `Task 1` performs $(\mathbf{B}_{1,1} + \mathbf{B}_{2,2})$, and so on. Similarly the `Task 10` performs the multiplication of the resulting matrices of the `Task 0` and the `Task 1`. Following this procedure we built 21 tasks including the multiplications required to compute $\mathbf{M}_i$ and the additions to compute $\mathbf{C}_{i,j}$.

3. *Data Dependencies.* The dependence graph for this application is depicted in the Figure: 7.1. Tasks 0 to 9 execute the additions and subtractions, tasks 10 to 16 execute the sub matrix multiplications, and finally tasks 17 to 20 execute the additions to compute the $C_{i,j}$ blocks.

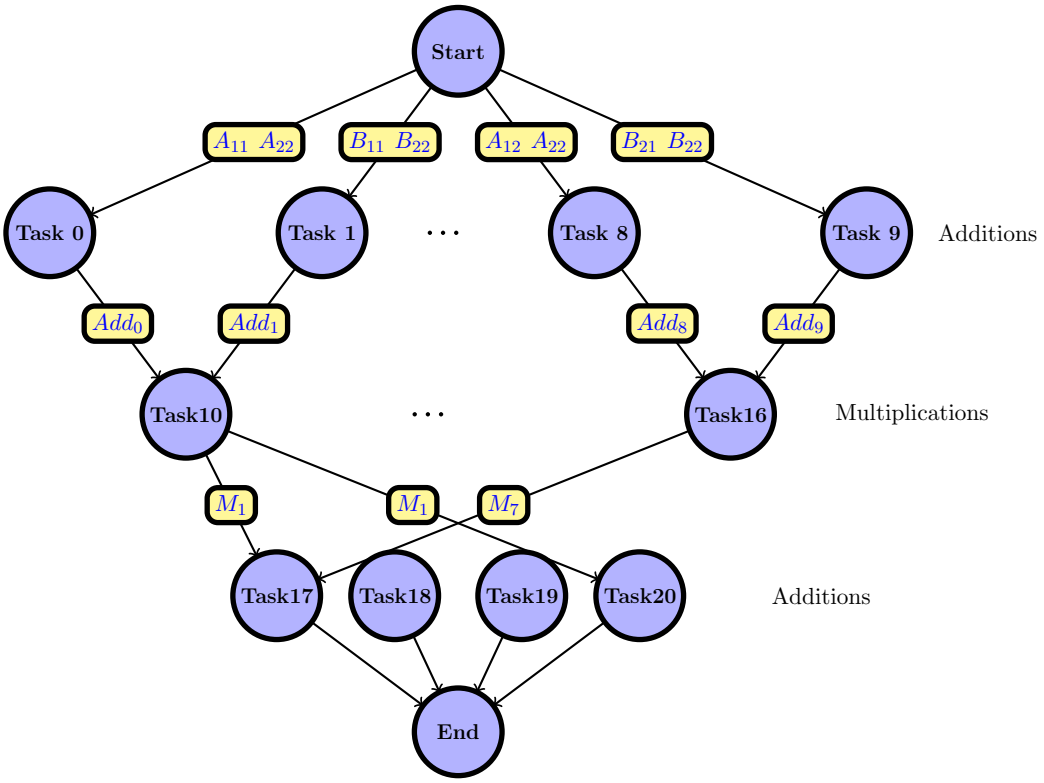


Figure 7.1: Task dependency graph for the Strassen's algorithm. Tasks 0 to 9 execute the additions and subtractions, tasks 10 to 16 execute the sub matrix multiplications, and finally tasks 17 to 20 execute the additions to compute the $C_{i,j}$ blocks.

4. *Scheduling.* Considering our knowledge about the number of tasks in the application as well as their data dependencies we used three scheduling techniques: HEFTMC, a round robin distribution and a manual configuration file.

5. *Implementation.* The initialization of the computing devices is automatically executed by the XSCALA environment eliminating several function calls like `clGetPlatformIDs`, `clGetDeviceIDs`, `clCreateContext`, and

`clCreateCommandQueue`, thus the first step in our implementation is the data distribution.

In the line 1 of Listing: 7.1 we show how to write the sub-matrix (A_{11}) in the `TRAY1` of the computing device assigned to the `task[0]`. The lines of code required to complete the same operation using MPI+OpenCL are depicted in the Listing: 7.2.

```
1 err|=XSCALA_WriteTray(task[0].ID, TRAY1,
      SIZE/2,  A11, MPI_COMM_WORLD);
```

Listing (7.1) A XSCALA data write.

```
1  err= clGetPlatformIDs(1, &PlatformID, NULL);
2  err =clGetDeviceIDs(PlatformID,
       CL_DEVICE_TYPE_ALL, 1, &device_id, NULL
       );
3
4  if (myRank == N) {
5      cl_context context = NULL;
6      context = clCreateContext( NULL, 1,
       device_id, NULL, NULL, &status);
7
8      cl_command_queue cmdQueue;
9
10     cmdQueue = clCreateCommandQueue(context,
        device_id, 0, &status);
11
12     cl_mem bufferA11;
13
14     bufferA00 = clCreateBuffer(context,
       CL_MEM_READ_ONLY, datasize, NULL,
        &status);
15
16     err|= clEnqueueWriteBuffer(cmdQueue,
       bufferA, CL_FALSE, 0, datasize, A11, 0,
        NULL, NULL);
```

Listing (7.2) MPI+OpenCL data write.

Figure 7.2: A comparison of the coding lines required for writing data on the memory of a device between the XSCALA framework (left) and the MPI+OpenCL approach (right).

Next we define the procedures of each task. In the line 2 of Listing: 7.3 we used the `XSCALA_SetProcedure` function call to set the "`Addition`" procedure in the "`task[0]`". The number of steps required to perform the same operation in OpenCL are depicted in Listing: 7.4.

```
2 err|=XSCALA_SetProcedure(MPI_COMM_WORLD,
    task[0].ID,srcPath,"Addition");
```

Listing (7.3) XSCALA procedure setup.

```
17     cl_program program =
       clCreateProgramWithSource( context, 1,
       (const char**) & srcPath, NULL, &err);
18
19     status = clBuildProgram(program, 1,
       device_id, NULL, NULL, NULL);
20
21     cl_kernel kernel = NULL;
22
23     kernel = clCreateKernel(program, "
       Addition", &err);
```

Listing (7.4) OpenCL kernel setup

Figure 7.3: A comparison of the function calls required for setting up a task in the XSCALA framework against OpenCL.

Similar calls are required to set the remaining multiplication and subtraction procedures. To harness the advantages of data parallelism the matrix multiplication procedure of the corresponding tasks is implemented using the tiled matrix multiplication approach described in Appendix: B that maximizes the use of shared memory on the device increasing the granularity of the problem.

Once the data is stored and the procedures have been defined, we can request the execution. The line 3 in Listing: 7.5 shows how to requests the execution of the kernel using the `XSCALA_ExecTask` function. The dimension of the work group are specified using the `globalDims` and `localDims` but unlike the OpenCL request shown in Listing: 7.6 the parameters for the procedure are passed using a simple "`%T`" notation.

```
3  err|=XSCALA_ExecTask(MPI_COMM_SELF, task
     [0].ID, Dims, globalDims, localDims,
     "%T %T %T",TRAY0,  TRAY1,  TRAY2);
```

Listing (7.5) XSCALA task execution request.

```
24     err = clSetKernelArg( kernel,
25         0, sizeof(cl_mem),  &bufferA00);
26     err |= clSetKernelArg( kernel, 1,
27         sizeof(cl_mem),  &bufferA11);
28     err |= clSetKernelArg( kernel, 2,
29         sizeof(cl_mem),  &bufferS00);
30
31     err = clEnqueueNDRangeKernel( cmdQueue,
32     kernel,  Dims, NULL, globalDims,
       localDims, 0, NULL,  NULL);
```

Listing (7.6) OpenCL kernel execution

Figure 7.4: A comparison of the function calls required for requesting the execution of a task in XSCALA against OpenCL.

Finally the `XSCALA_SendRecv` function call in line 4 of Listing: 7.7 illustrates the simplification in the number of operations required to complete a data transfer in comparison to the MPI+OpenCL calls depicted in Listing: 7.8.

```
4  XSCALA_SendRecv(task[12].ID, TRAY0, task
     [18].ID, TRAY1, NS, MPI_FLOAT);
```

Listing (7.7) XSCALA Inter task data transfer

```
34  if (myRank == SENDER) {
35      clEnqueueReadBuffer(cmdQueue, bufferC,
        CL_TRUE, 0,  datasize,  S0,  0, NULL,
        NULL);
36
37      err = MPI_Send(&outmsg, 1, MPI_CHAR,
        dest, tag, MPI_COMM_WORLD);
38  }
39
40  else if (myRank == RECEIVER) {
41      err = MPI_Recv(&inmsg, 1, MPI_CHAR,
        source, tag, MPI_COMM_WORLD, &Status);
42
43      err|= clEnqueueWriteBuffer( cmdQueue,
        bufferA, CL_FALSE, 0, datasize, A00, 0,
         NULL, NULL);
44  }
```

Listing (7.8) OpenCL + MPI data transfer

Figure 7.5: Comparison of the function calls required for transferring data in XSCALA against OpenCL+MPI.

Experimentation and Results

The experiments were performed as follows: We implemented the algorithm using OpenMP executing this program in the "`CPU_0`" of platform C. Another implementation using CUDA was executed in the "`GPU_0`" of the platform C. Finally we implemented the algorithm with XSCALA using the devices in the platform C.

The objective of these implementations consists in bring an insight of the performance achieved using the traditional programming tools against XSCALA, and to verify the level of scalability achieved.

The results of this experiment are shown in the Figure: 7.6, where each bar represents the time required to complete the execution with the given tools, times is measured in seconds and plotted using a semi log base 10 scale to ease the visualization of the results.

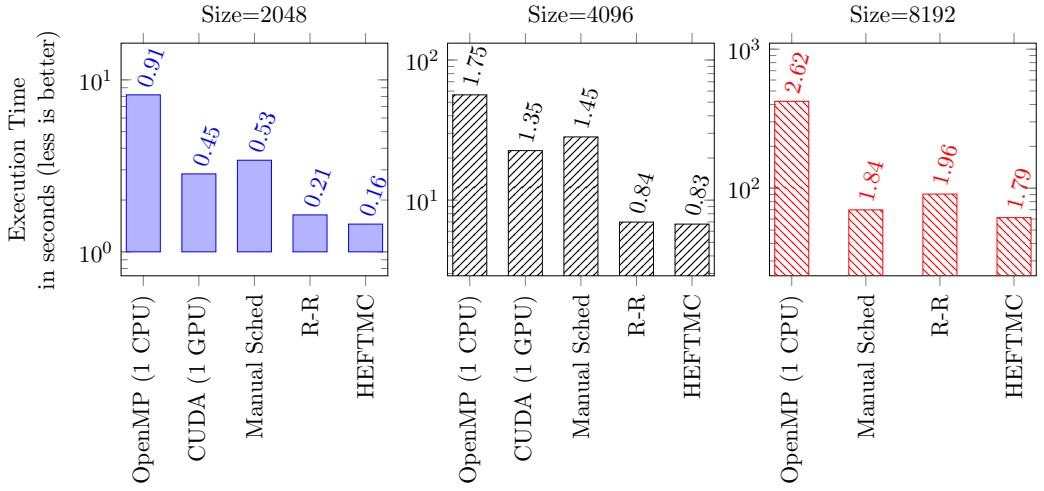


Figure 7.6: Comparison of the execution time for three different sizes of SGEMM using single precision floating point. Execution times (secs) are plotted in a semi log plot base 10.

Next we compared HEFTMC against other two strategies: round robin (R-R) and the manual configuration file depicted in the Figure: 7.7. The configuration file maps the tasks 0 to 9 to the device `CPU 0` in the node 0, the task 10 to the device `CPU 0` in the node 0, the task 11 to the device `GPU 0` in the node 1 and so on.

We measured the time required to complete each task in the application for the three different scheduling algorithms using squared matrices of size 2048, The results of the measures for each scheduling strategy are shown in the Gantt diagrams plotted in the Figure: 7.8. Each rectangle in the plot represents the initialization and finalization time required by each task and the separation between the rectangles represents the communication overhead for example, in manual scheduling the former 10 tasks where

```
# TaskID    Node    Device
  0-9        0      CPU 0
  10         0      CPU 0
  11         1      GPU 0
  12         0      CPU 1
  13         1      GPU 1
  14         0      CPU 2
  15         1      GPU 2
  16         0      GPU 3
  17-20      0      CPU 0
# End config file
```

Figure 7.7: Task configuration file for the Strassen's application application.

mapped in the `"CPU_0"` delaying the execution of the dependent tasks. The plot of the R-R scheduling makes another distribution where in addition to the distribution of the multiplication task the additions are also distributed reducing the time required to complete the dependencies.

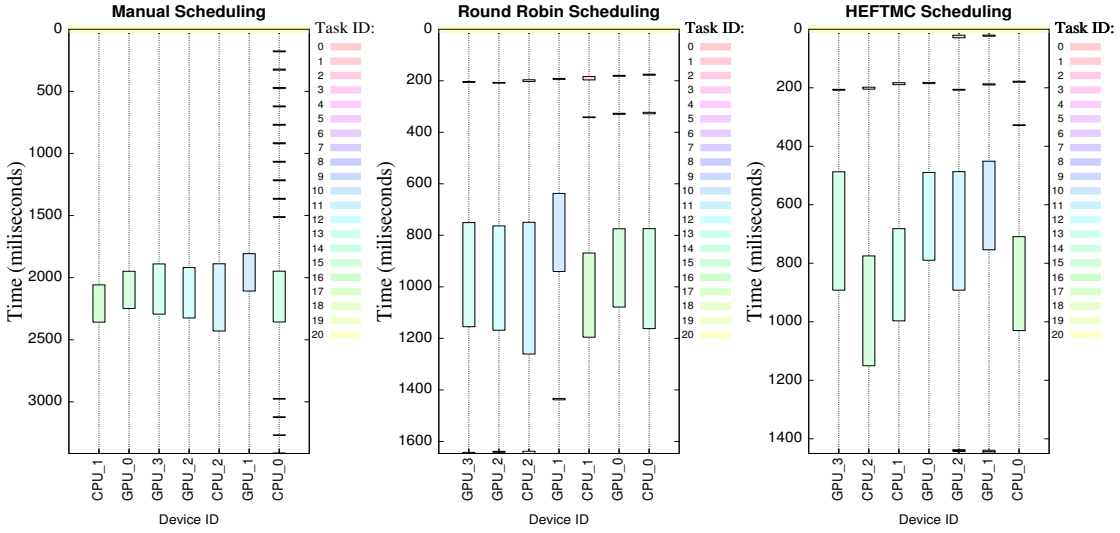


Figure 7.8: Three scheduling alternatives for a problem of size 2048. The rectangles shown the actual execution time of a task, and the dashed lines shown the overhead for communication and initialization of the tasks. In the HEFTMC scheduling the tasks were allocated closer to their dependencies minimizing the data transfers and achieving the best performance.

In the first plot we have the performance of the "*Manual scheduling*" strategy that only distributes the execution of the matrix multiplications and requires 3410 miliseconds to complete the execution. In the second plot the task were distributed using a Round Robin distribution requiring 1646 miliseconds to complete. Finally, in the third execution we used our HEFTMC algorithm requiring only 1450 miliseconds to complete the execution.

Discussion

In our first experiment varying the size of the matrices from 2048, 4096 and 8192 we compared the performance against the OpenMP and CUDA implementation. From the results of the experiments we can appreciate that workload distribution given by HEFTMC improves the performance of the application achieving a speed up of 5.64x, 8.35x, and 6.86x with respect to the OpenMP implementation.

In the case of matrices with size 8192 using single precision floating point format the multiplication requires of 768 MB of memory space and our CUDA implementation for a single device could not be executed due to the lack of memory space demonstrating the importance of the scalability that XSCALA can provide.

In our second experiment where we analyzed the performance of the schedulers we could appreciate the advantages of the static scheduling approach.

7.3 Scalable Linear Algebra with XSCALA

In this section we present a task-based implementation of two classical subroutines of the standard basic linear algebra subroutines (BLAS) package [Lawson et al., 1979]. In particular we present our scalable implementation of the SAXPY and SGEMM subroutines described in the Table: 7.4.

Problem	Description	BLAS Level	Problem sizes
SAXPY	Task based implementation of the single precision scalar vector multiplication with vector vector addition	1	$2^{22} - 2^{28}$
SGEMM	Task based implementation of the single precision general matrix matrix multiplication problem	3	$2^{10} - 2^{14}$

Table 7.4: BLAS applications.

The objective of our SAXPY and SGEMM applications is to show how to use the entity data types to perform the distribution of big files between multiple tasks easing the construction of scalable applications.

7.3.1 SAXPY

Introduction

SAXPY stands for "***S**ingle precision **A**lpha **X** **P**lus **Y***", and consists of the multiplication of a scalar α by a vector X followed by the addition of another vector Y using single precision floating point format.

The blocks of entities are scattered between the tasks (instead of between the devices) using the XSCALA's scatter operation The procedure assigned to each task performs the scalar-vector multiplication and vector-vector addition on the set of entities assigned to each task, and then the results of each task are gathered in a output file.

Development

1. *Entities Definition.* In the traditional implementation of SAXPY the vectors X and Y are divided and scattered according to the number of devices available on each node, then the blocks are divided again to store a portion of data in the memory of each device. This approach inhibits the scalability of the application because the scatter operation assumes that the number of devices per node is fix.

   To overcome this limitation our task-based implementation defines an entity as a data structure of three numbers, one for each entry of the vectors X, Y and Z as is depicted in the Figure: 7.9.

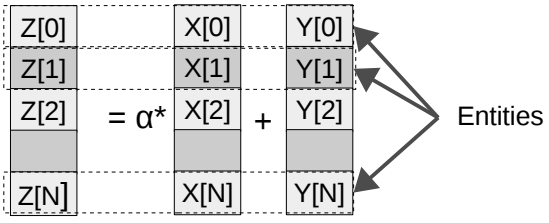


Figure 7.9: The task-based implementation of SAXPY.

2. *Tasks Definition.*

   The procedure assigned to each task is shown in the Listing: 7.9and works as follows: the line 3 gets the ID of thread an line 7 performs the multiplication of the $\alpha \cdot x + y$ operation in each entity.

3. *Data Dependencies.* The dependency graph for this application is depicted in the Figure: 7.10 where each tasks waits the block of entities assigned from the scatter operation and once they are completed proceed to the gather operation.

```
1  __kernel void vecAdd(__global entities * entity,const int alpha, const int n){
2   //Get our global thread ID
3      int ID = get_global_id(0);
4      //Make sure we do not go out of bounds
5      if (id < n)
6          entity[ID].z = alpha*entity[ID].x + entity[ID].y;
7  }
```

Listing 7.9: Procedure of the tasks in SAXPY

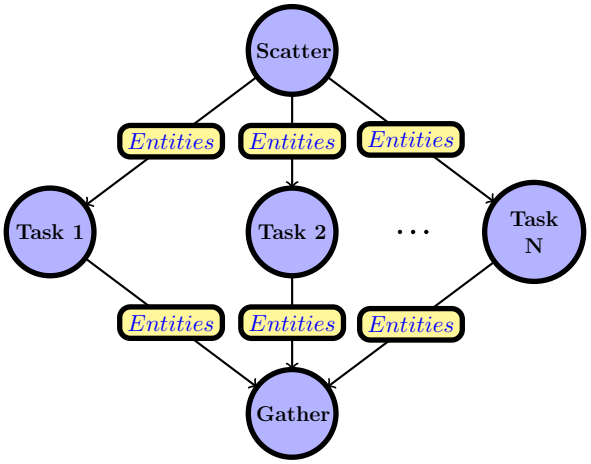


Figure 7.10: Workflow for the SAXPY subroutine.

4. *Scheduling.* We used configuration files with 50% of the workload for the CPUs and 50% to the GPUs when both devices were used.

5. *Implementation.* Listing: 7.10 shows the XSCALA implementation of SAXPY . Line 5 creates the entity data structure, in line 8 we query the number of tasks in the system and then the *ePerTask* variable in line 9 computes the number of entities that will be processed on each task (assuming it is a multiple otherwise might be filled with zeros). Line 11 performs the scatter of the entities from the data file to the tasks memory. Line 19 sets the procedure and line 20 requests the execution of the task. finally line 22 performs a synchronization point before the gather operation of the line 23.

Experiments and Results

The experiments were performed as follows: We generated multiple data files storing the vectors in the format of entities, the size of the vectors and the size of the files generated is summarized in the Table: 7.5. In this case we the problem size refers to the number of vector entries.

```
#define TRAY0 0
int err;
                    ⋮
MPI_Datatype entity;
XSCALA_CommitEntity(blockcount, blocklen,  disp,  basictypes, &entity);

int numTasks;
err=XSCALA_GetNumTasks(&numTasks, MPI_COMM_WORLD);
int ePerTask=(int)VectorSize/numTasks; //entities per task

err=XSCALA_Scatter("dataFile.dat", entity, TRAY0, MPI_COMM_WORLD);

int Dims=1;
size_t globalDims[]={((ePerTask-1)/64+1)*64}; /*to ensure a WG multiple of 64.*/
size_t localDims[]={64};

int alpha=2;
for(tskIdx=0;tskIdx<numTasks;tskIdx++){
    err=XSCALA_SetProcedure(MPI_COMM_WORLD, tskIdx, "vecAdd.cl","vecAdd");
    err=XSCALA_ExecTask(MPI_COMM_SELF,tskIdx,Dims,globalDims,localDims,"%T %d
                    %d",0,alpha,ePerTask);
}
err=XSCALA_WaitAllTasks(MPI_COMM_WORLD);
XSCALA_Gather(TRAY0, entity, "resultsFile.dat",  MPI_COMM_WORLD);
                    ⋮
```

Listing 7.10: Task based implementation of SAXPY

Problem Size	File Size
2^{22}	48 MB
2^{23}	192 MB
2^{24}	384 MB
2^{27}	1.5 GB
2^{28}	3 GB

Table 7.5: SAXPY file size.

We tested this application in the experimental platform C using manual scheduling and varying the number of hosts and the number of tasks on each execution. The results of the executions are summarized in Table: 7.6. The first three columns describe the execution environment for example 2 nodes using CPU+GPU yields four computing devices for the execution, the next columns define the problem size and the *megaflops* achieved in each execution.

Figure: 7.11 shows the performance obtained in the execution of the SAXPY subroutine using 1,2 or 4 nodes. In the Figure: 7.11a we appreciate a performance degradation of nearly 0.5x with respect to the single GPU execution using one node and two devices and even gets worst on the case of four nodes yielding a 0.24x. This degradation can be explained because the granularity of this problem remains as $O(1)\frac{ops}{access}$ regardless of the size of the problem the performance of the application is bounded by the time

			PROBLEM SIZE				
	# NODES	Node Config.	2^{22}	2^{23}	2^{24}	2^{27}	2^{28}
CONFIGURATION	1 Node	CPU	103	160	174	175.740	177.497
		GPU	295	458.252	498.35	-	-
		CPU+GPU	150	233.01	253.398	-	-
	2 Nodes	CPU	72.100	112	121.8	123.018	124.248
		GPU	206.5	320.777	348.845	-	-
		CPU+GPU	105	163.107	177.379	179.152	-
	4 Nodes	CPU	50.470	78.4	85.260	86.113	86.974
		GPU	144.550	224.544	244.191	246.633	-
		CPU+GPU	73.5	114.175	124.165	275.026	286.01

Table 7.6: Results of the execution of SAXPY. All the results presented in this table are measured in megaflops. The "-" symbol means that the application could not be executed due to lack of memory space on the devices.

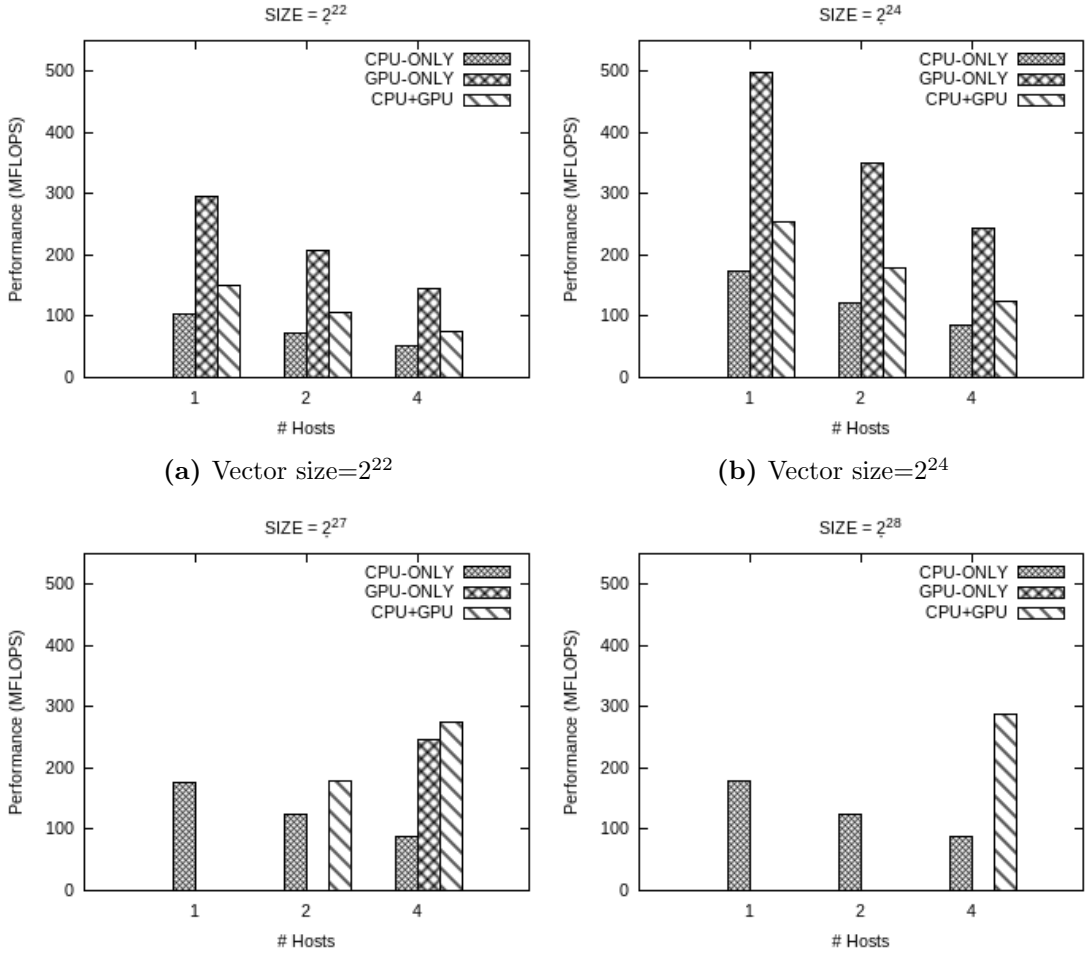
required for data distribution resulting in a poor level of scalability as more hardware resources are added specially when we add more nodes for example, we can appreciate a the peak performance achieved for the problem size 2^{24} using only one GPU in the Figure: 7.11b, and adding more resources do not improves the performance.

In the Figure: 7.11c we can appreciate an increase in performance with respect to the CPU-ONLY execution. Here we can appreciate that some executions using the GPU-ONLY configuration could not be completed due to the lack of memory space on the devices highlighting the importance of construct scalable implementations. This effect becomes more evident in the Figure: 7.11d where the use of GPUs ONLY is unable to complete any execution but the combination of all the devices completes and enhances the performance with respect to the use of a single CPU.

Discussion

Following the task programming model we were able to construct a scalable implementation of SAXPY where the data can be divided among several tasks regardless of the vector size, the number and the location of the computing devices without requiring modifications to the application code.

In spite of this problem we can appreciate the adaptability of the application for shared and distributed memory environments as well as the possibility of solving larger problems for example for the size 2^{28} that could not be solved using a single device due to the memory constraints. As a conclusion we have that for small sizes the best approach



(a) Vector size=2^{22} **(b)** Vector size=2^{24}

(c) Vector size=2^{27} **(d)** Vector size=2^{28}

Figure 7.11: Performance of the SAXPY application implemented with XSCALA for multiple problem sizes and multiple devices of the platform C. The performance is measured in *megaflops* (more is better) and zero megaflops means that the application could not be executed with that combination due to the lack of memory space.

consists in execute the application in a single device and use our scalable approach for larger sizes.

7.3.2 SGEMM

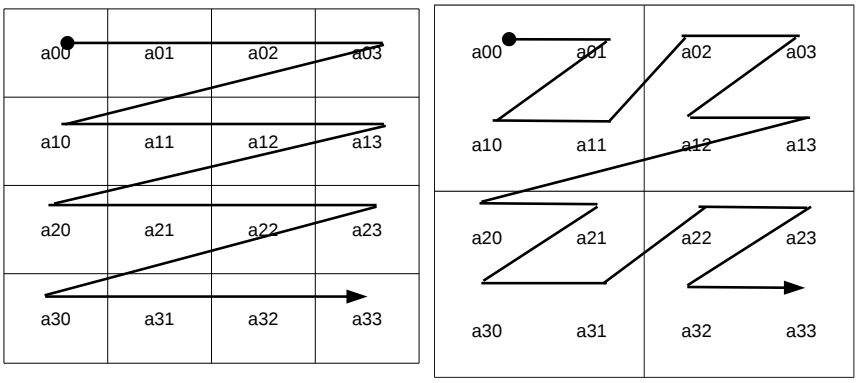
Introduction

SGEMM stands for "***S**ingle precision **GE**neral **M**atrix **M**ultiplication*". This application implements a task-based version of the SGEMM problem with the objective of obtain a scalable implementation to solve bigger problems preserving the structure of the code when more hardware resources are added.

The major restriction to achieve scalability in this application is the problem of data dependencies to perform the inner product between rows of A and columns of B. In order to address this restriction we will divide the matrix in blocks and use task parallelism to perform multiple parallel block matrix multiplications.

Development

1. *Entities Definition.*We define the entities for the block matrix multiplication as submatrices of the original matrices A and B respectively. To build such blocks we need modify the row major form of storage shown in Figure: 7.12a to the block major form shown in Figure: 7.12b.



(a) Row major form. **(b)** Block major form.

Figure 7.12: Entites are defined as sub-matrices of matrix A.

Once the matrices are stored in the appropriate format we can proceed to perform matrix multiplications using one task per block.

2. *Tasks Definition.* To harness the advantages of data parallelism the procedure of each task for the multiplication of two blocks is implemented using the tiled matrix multiplication approach described in Appendix: B.

3. *Data Dependencies.* Each block matrix is stored in one tray of the task where it is required and is transferred at each step of the block matrix multiplication. In this case the blocks are initially distributed according to the labels shown in the Figure: 7.13, and are transferred at each step as follows: The blocks of A are transferred to the task pointed by the dashed arrow and the blocks of B are transferred to the task pointed by the continuous arrow. for example in the step 0 the $Task0$ performs the multiplication $A0 \cdot B0$, in the step one $Task0$ gets the blocks $A1$ and $B4$ from its right and button neighbors respectively and then performs the multiplication $A1 \cdot B4$. This process is executed up to the completion of all blocks.

Figure 7.13: Data interchange pattern in block matrix multiplication.

This data dependency pattern has the advantage to transfer small chunks of data using different memory hierarchies at a time avoiding the saturation of the inter-node communication channel.

4. *Scheduling.* For this application we used manual scheduling defining a configuration file for each execution.

5. *Implementation.* In the Listing: 7.11 we present part of the implementation of the block matrix multiplication using four blocks per matrix. In the line 3 of this code we request the multiplication of two block matrices, this multiplication is embedded in the inner for loop causing that all the execution requests are executed in parallel using different devices. In lines 12 and 14 we transfer the blocks of matrices A and B between the tasks following the pattern described before and finally line 18 performs a synchronization among all tasks before to proceed with the next step of the outer loop.

Experimentation and Results

The experiments were performed using the experimental platforms B and C varying the sizes of the matrices, the number of devices and the number of hosts. Here the problem size refers to the number of rows and columns in each matrix.

```
for(step = 0; step < sqrt(numTasks); step++) {
 for (taskIdx = 0; taskIdx < numTasks; taskIdx++) {
  err |= XSCALA_ExecTask(MPI_COMM_SELF, taskIdx, Dims, globalDims,
                         localDims,"%T %T %T %d %d ", 3*(step%2), 3*(step%2)+1,
                         2, rPBlock, cPBlock);
   srcATray =3*(step%2);
   destATray=3*((step+1)%2);

   srcBTray =3*(step%2)+1;
   destBTray=3*((step+1)%2)+1;

   err |= XSCALA_SendRecv(myRight[taskIdx], srcATray ,taskIdx , destATray, 1,
                          Aentity, MPI_COMM_WORLD );
   err |= XSCALA_SendRecv(myDown[taskIdx], srcBTray ,taskIdx , destBTray, 1,
                         Bentity, MPI_COMM_WORLD );

 }
err |= XSCALA_WaitAllTasks(MPI_COMM_WORLD );
}
```

Listing 7.11: Task based implementation of SGEMM with XSCALA

In our first experiment we used squared matrices of sizes from 256 to 4096. The results of the execution of SGEMM employing 1,2 and 4 of the GPUs of the platform C are depicted in the Figure: 7.14.

From these results we can see that the strategy of combine several GPUs is convenient when the size of the problem is enough to hide the communication overhead for example, we can see an important increase in performance when we used the four GPU cards for a problem of size of 4096 but using two GPUs is the best approach for sizes of 2048 and below.

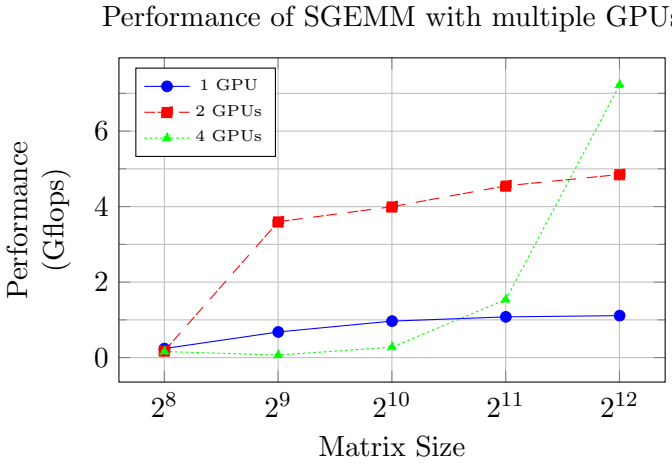


Figure 7.14: Performance of SGEMM varying the number of GPUs on the platform C.

In our second experiment we used multiple combinations of GPUs of the platform B. The results are depicted in the Figure: 7.15. In this case we created five scheduling

configuration files to dispatch four tasks with the follow configurations: the first three executions assigned 100% of the workload to each one of the GPUs, then we used two GPU cards with 50% of the workload per device and finally we used the tree GPU cards assigning 50% of the workload to the TESLA GPU, 25% to the QUADRO, and 25% to the GTX card.

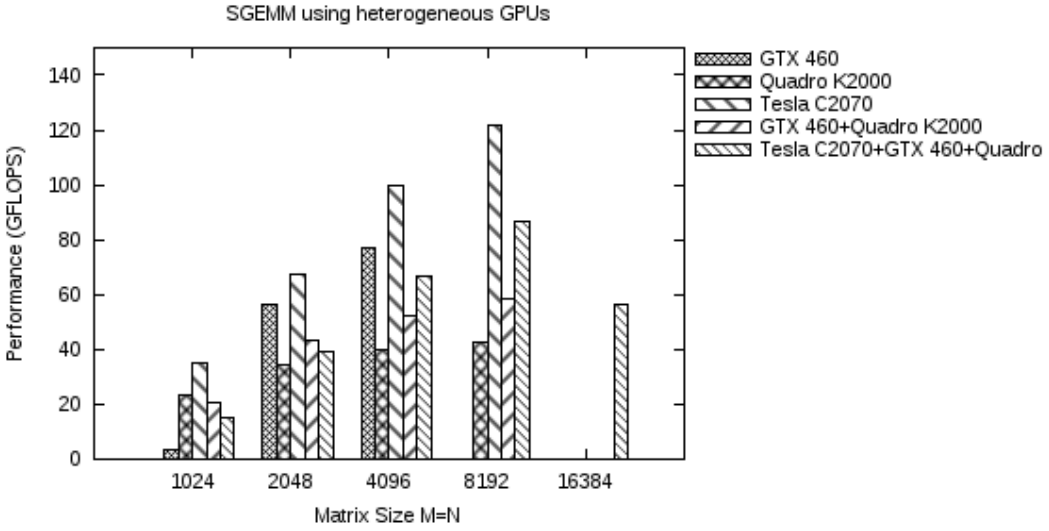


Figure 7.15: Performance in the execution of SGEMM using multiple GPU accelerators. 0 Gflops means that the application could not be executed due to lack of memory space.

In this case we see that as we increase the size of the problem beyond 4096 the GTX becomes unable to complete the execution due to the lack of memory space, and for a size of 16384 the only option is the use the combination of all the GPUs.

Finally in the Figure: 7.16 are depicted the results of the executions using several combinations on the platform C.

In this case as we increase the size of the problem a single node becomes unable to perform the multiplication due to the lack of memory space, however the application can use distributed memory resources to sparse the data without any modification enabling the execution of the multiplication for a size of 8192 with optimal performance.

Discussion

From the results of our experiments we see that the granularity of this problem $O(\frac{\text{Block Size}^3}{\text{Transfer}})$ enables to achieve weak scalability resulting in a better performance when more resources are used.

The SGEMM application is able to execute successfully in multiple environments using the XSCALA framework which is able to solve bigger problems by collecting memory

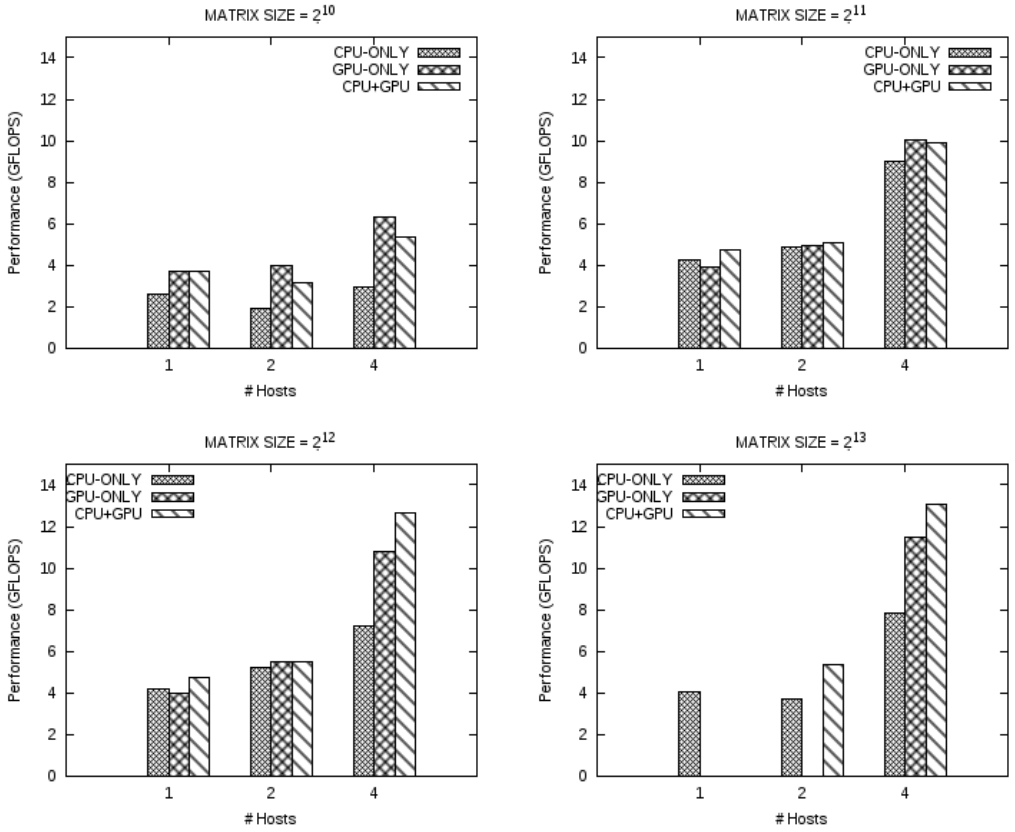


Figure 7.16: Performance in the execution of SGEMM using multiple sizes and multiple GPUs of the platform C.

resources form several GPUs to sparse the data without requiring modification to the source code demonstrating the scalability of the application when is designed with a task-based approach.

7.4 The N-Body Simulation Problem.

Introduction

This application is a simulation of the interaction of N bodies subject to gravity forces. The objective of the algorithm consists in keep track of the position and velocity of each body (atoms, stars, etc) in the system. On its simplest form the N-Body problem will compute the forces between each pair of bodies to determine the position of the particle in the next interval of time. The Algorithm: 10 implements the N-Body simulation and work as follows: Let the vector $\overrightarrow{r_(t)}$ to represent the position of the i^{th} body at any given time, $\overrightarrow{a_i(t)}$ to represent the acceleration and r_{ij} the distance between two bodies. In line 5 we compute the forces acting on each body using and line 6 we compute the

acceleration of the body. The total force acting on each body is computed using the Equation: 7.4.

$$\overrightarrow{F}(i) = Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \overrightarrow{r}_{ij}}{(\|\overrightarrow{r}^2_{ij}\| + \epsilon^2)^{3/2}} \tag{7.4}$$

Line 7 and 8 updates the position and velocity of the body using classical mechanics equations, and finally line 11 forwards the time step. This version of the N-Body algorithm has a complexity $O(N^2)$ with N the number of bodies.

input: initial time, final time,
G = gravity constant,
ϵ = softening constant,
$r_i(t_0)$= initial body set configuration,
$v_i(t_0)$= initial body set velocity.

Output: Final body set configuration

1: t← initial time;
2: **while** t ≤ final time **do**
3: **for** each body i in the system **do**
4: **for** each other body j **do**
5: $\overrightarrow{F_i} = Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j \overrightarrow{r}_{ij}}{(\|\overrightarrow{r}^2_{ij}\| + \epsilon^2)^{3/2}}$ /* Force Calculation */
6: $\overrightarrow{a_i} = \frac{\overrightarrow{F_i}}{m_i}$ /* Acceleration Calculation */
7: **end for**
8: $\overrightarrow{r_i}(t+1) = \overrightarrow{r_i}(t) + \overrightarrow{v_i}(t) * \Delta t + \frac{1}{2}\overrightarrow{a_i}(t) * (\Delta t)^2$ /*update positions*/
9: $\overrightarrow{v_i}(t+1) = \overrightarrow{v_i}(t) + \overrightarrow{a_i}(t) * \Delta t$ /*update positions*/
10: **end for**
11: $t = t + \Delta t$; /* Forward Time*/
12: **end while**

Algorithm 10: N-Body Simulation.

Development

1. *Entities Definition.* For the N-Body problem we define an entity as each body in the system, the entity is composed by the position, the weight, and the velocity of the particle using three spatial dimensions to represent the position and three for the velocity respectively.

2. *Tasks Definition.* The task-based implementation of N-Body consists in dividing the data file having the initial position of the bodies among the tasks assigning the same number of bodies per task. To compute the total forces acting on the

body the procedure must compute the forces due to the interaction with the other bodies in the task and then with the other bodies in other tasks.

3. *Data Dependencies.* To compute the forces due to the interaction with bodies allocated on other tasks we divide the computation of the forces in multiple steps performing a left shifting of entities at each step creating the dependency graph depicted in the Figure: 7.17.

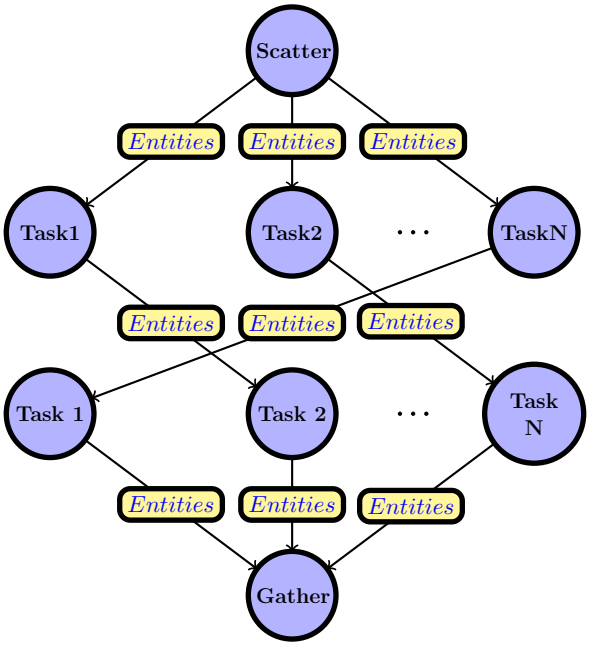


Figure 7.17: Workflow of the NBody Application.

4. *Scheduling.* For this application we used manual scheduling defining a configuration file on each execution.

5. *Implementation.* Part of the code for the implementation of the N-Body problem with XSCALA is shown in Listing: 7.12. In line 4 we set the definition of the entities aforementioned. Line 9 gets the number of tasks defined in the system, and line 11 scatters the entities to the tasks. In line 14 we set the procedure for each task. In lines 17 to 25 we perform the computation of forces between each pair of entities, and finally line 26 works as a synchronization point among all tasks. This process is repeated until we reach the end of the simulation.

Experimentation and Results

The experiments were performed using the platforms **A** and **B** varying the number of bodies and the number of devices on each case. In this application the problem size refers to the number of bodies in the simulation.

```
#define mymod(n,m) ((n % m) + m) % m;
#define TRAY0 0
#define TRAY1 1
XSCALA_CommitEntity(blockcount, blocklen, disp,
                        basictypes, &entityType);
int err; // To catch error messages.

int numTasks; /*Get the number of tasks*/
err = XSCALA_GetNumTasks(&numTasks, MPI_COMM_WORLD);
        /*We perform data distribution among the tasks.*/
err = XSCALA_Scatter("dataFile.dat", entityType, TRAY0, MPI_COMM_WORLD);

        /*Create the procedure.*/
err |=XSCALA_SetProcedure(MPI_COMM_WORLD, "NBodyExt.cl", "computeForces");

        /*Finally perfom the execution.*/
for (step = 0; step < numTasks; step++) {
    for (taskIdx = 0; taskIdx < numTasks; taskIdx++) {
          int srcTask = mymod((step+taskIdx-numTasks),numTasks);
          err |= XSCALA_SendRecv(srcTask, TRAY0, taskIdx, TRAY1,
                                 ePerTask, entityType, MPI_COMM_WORLD );
          err |= XSCALA_ExecKernel(MPI_COMM_SELF, taskIdx, Dims, globalDims,
                         localDims, "%T, %T, %d ,%f ,%f, %d ",TRAY0 , TRAY1,
                         numBodies,0.0005, 0.01, numTasks);
 }
 err |= XSCALA_WaitAllTasks(MPI_COMM_WORLD);
}
```

Listing 7.12: N-Body Initialization

In Figure: 7.18 we depict the results of three executions of the application measured in gigaflops (Gflops) in the platform A.

We created three scheduling configuration files to dispatch four tasks with the follow configurations: In the first case we used the CPU only, in the second we used the GPU only and finally we assigned 75% of the workload to the GPU and 25% in the CPU.

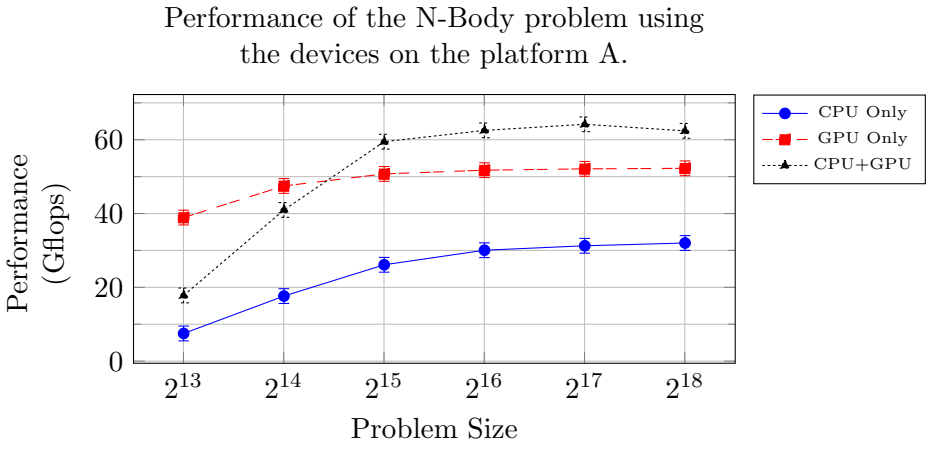


Figure 7.18: Performance of the N-Body problem using the CPU and the GPU in the platform A.

In the Figure: 7.19 are depicted the results of the execution of the n-body problem using multiple GPU cards in the platform B.

Here we created five scheduling configuration files to dispatch eight tasks with the follow configurations: the first three executions assigned 100% of the workload to each one of the GPUs, then we used two GPU cards with 50% of the workload per device and finally we used the tree GPU cards assigning 50% of the workload to the TESLA GPU, 25% to the QUADRO, and 25% to the GTX card.

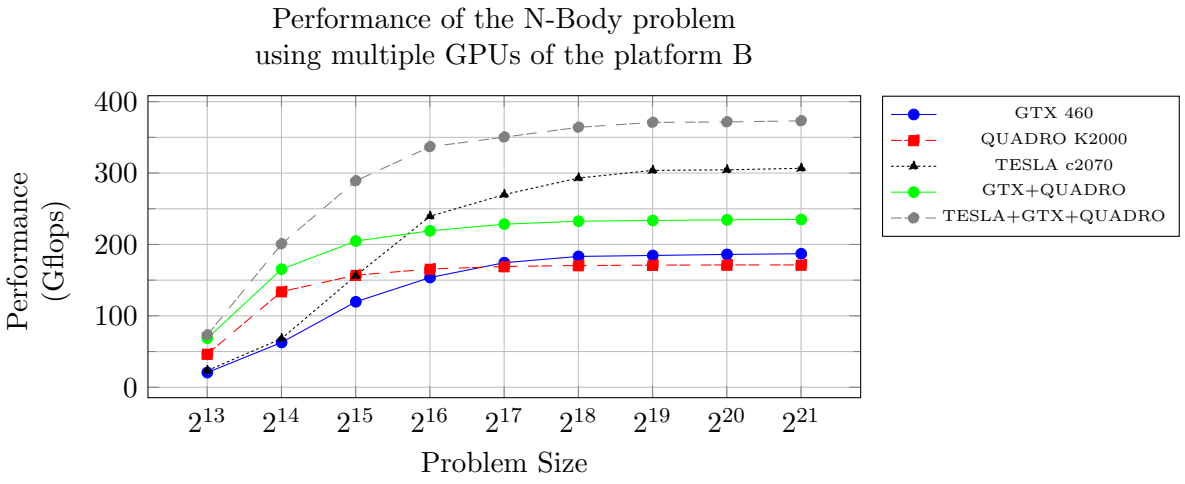


Figure 7.19: Performance of the N-Body problem using multiple GPU accelerators in platform B.

Discussion

From the results of execution in the platform A we appreciate that when the size of the problem is big enough to hide the communications overhead we can obtain an increase in performance achieving a peek performance of 65 *Gflops* using both devices against the 50 *Gflops* achieved when we use only the GPU. The granularity of this problem is $O(\frac{\text{ePerTask}^2}{\text{Transfer}})$ enables to achieve weak scalability.

The use of multiple GPU cards in the platform B results in a peak performance of nearly 400 Gflops, improving the best performance achieved using only the Tesla c2070 GPU.

Chapter 8

Conclusions and Future Work

The programming of hybrid heterogeneous cluster computers has high relevance not only for taking advantage of current systems but for the development of applications adaptable to future computing systems that will integrate several computing architectures in a single chip.

It has been seen that the speed in the production of new hardware is faster than the time required for the development of software able to harness the new hardware. In that sense the task programming model is a promising approach for developing applications that can be executed in future generations of computer systems having more complex heterogeneous architectures.

The flexibility of the task programming model has become more relevant encouraging novel research efforts to create tools that can support it. Several works found in the literature used the task approach implicitly or explicitly but staying in the scope of shared memory architectures. Our work extends the scope of the task programming model to reach both: shared and distributed memory environments.

The design and implementation of the XSCALA framework accomplishes the goal of generate portable applications able to be executed in several environments and scalable as more hardware resources are provided, as was demonstrated by the experimental applications that were implemented.

Evaluating the ease of programming achieved with XSCALA is not straight given that the terms "easy" and "difficult" are subjective. In spite of that we argue that the XSCALA API fulfills the objective of ease programming based in the fact that the hardware abstraction provided by our task model provides transparent management of heterogeneous computing resources.

Even though the number of lines of code required to implement an applications do not represents a formal measure of the complexity intuitively reflects the complexity for the implementation and the level of hardware abstraction provided by the development tools for example, a synchronization operation between tasks in XSCALA is significantly easier than dealing with mutexes, semaphores, and conditional variables used in the synchronization of threads.

One of the major concerns in executing tasks in H/H systems is the correctness of the executions. The middleware layer implements the algorithms required to complete the execution of applications as efficient as possible but foremost ensuring the correct order in the execution keeping in mind the global objective of this work: ease programming.

Given the complexity of the scheduling problem we known that there not exists a single algorithm that can provide optimal solutions for all the possible scenarios. The use of static scheduling heuristics has been deeply studied but they are seldom implemented due to the considerable amount of data and parameters required for their execution. In spite of these difficulties the use of static scheduling is still an attractive approach due to the possibility of planning the distribution of tasks and enhance the utilization of scarce resources like the memory space. On the other hand dynamic scheduling has the advantage of perform work distribution even when the number of task can change along the execution of the application enabling the integration of new tasks at run time. That are the major reasons why we decided to implement both: static and dynamic scheduling strategies in XSCALA.

An additional advantage that we achieved with our implementation of the framework is that hereinafter XSCALA becomes an experimental platform to test scheduling algorithms using real execution environments and not only simulations.

Memory management is another topic that was carefully considered in this work. The most common scenario in H/H systems is a small set of "fast" devices with limited amounts of memory and a larger set of "slow" devices having bigger memory spaces. Our contributions in this respect were the construction of a formal definition of the problem and the development of algorithms that can find a trade-off between performance and maximization of the memory occupancy.

In spite of the advantages of the task programming model, not all the applications can be implemented using the task model model for example, recursive algorithms can be readily expressed using structured programming models.

Performance and scalability are two major concerns in the development of applications however they can not be always guaranteed, sometimes distributing the workload in multiple nodes might result in a performance degradation due to the communications

overhead. A fundamental criterion for determining whether an application will improve its performance when more hardware is aggregated is the granularity of the problem. Applications having fine granularity will perform much better when they are implemented using vendor specific programming tools like CUDA or TBB. Another criteria is the size of the problem, however determining when the problem is big enough to hide communications overhead is also a complex task.

Defining rules to determine whether the granularity of the applications is enough to be executed in H/H systems or the size of the problem is big enough are crucial for the success of the XSCALA framework given that a programming effort to implement an application with XSCALA only to discover that it will never run better than in a single device with CUDA could result very disappointing.

Future Work

The architecture of XSCALA enables the integration of more features and improvements to the current modules. Undoubtedly scheduling is by itself an important direction for new research and multiple heuristics can be implemented and tested, in particular the use of preemptive scheduling techniques is a promising approach since there exists some variations of the scheduling problem that can be solved in optimal way using polynomial time algorithms when preemption is allowed, however the challenge to achieve preemption would be the implementation of mechanisms to migrate tasks at runtime.

Another important service that can be implemented in a future work is the fault tolerance service. As the computing system gets bigger the possibility that something fails also increases, mainly because the communication channels or certain nodes might fail. Fault tolerance requires the design of mechanisms to take snapshots of the state of each tasks and rollback the system to a previous state. Aside of this the problem of fault tolerance requires the implementation of consensus algorithms to determine the origin of the failures and the integration of stochastic models to determine the most probable points of failure and implement effective mechanisms for task replication. Fault tolerance is undoubtedly an important direction for new research in this work.

Dynamic device integration is another interesting direction of research. The complexity of some execution environments having multiple-costs makes us wonder how to implement a mechanism to use certain devices only when the application reaches a predefined performance threshold.

The integration of XSCALA into GRID computing environments also represents an important direction to work. GRID environments consists of multiple clusters of computers

geographically distributed connected through low bandwidth global area networks. For this type of environment, scheduling must consider a harder set of rules including administrative permissions for example the maximal compute capability that can be used or the number of granted devices.

Finally a service useful for programming applications is deadlock detection. There is nothing more frustrating than spending several hours executing an application only to discover that it has not advanced at all and in fact the whole system is blocked. The task programming model has the advantage of defining its data dependencies in advance enabling the implementation of deadlock detectors that can find possible sources for deadlock.

Appendix A

XSCALA Code Samples

In order to give a brief introduction to the use of the XSCALA API we present the full code for the implementation of the token ring application. Here we demonstrate the use of basic operations like declaring a task, setting procedures and defining data dependencies.

A.1 Static Token Ring.

In this application the token consists of an integer that must be passed forward between the tasks. We define six tasks labeled from 0 to 5 and the procedure of each task consists in let 10 threads adding one, one at a time, to the value of the token. A figure of the dependency graph of this application is depicted in the Figure: A.1.

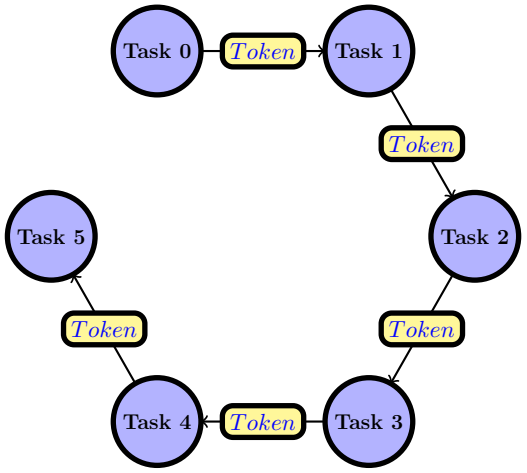


Figure A.1: Token Ring Dependency Graph.

Given that the number of tasks remains fixed we can use the manual scheduling strategy mapping two tasks per node one in the CPU and the second in the GPU. The final result of the operation and the architecture of the execution environment is depicted in the figure A.2,

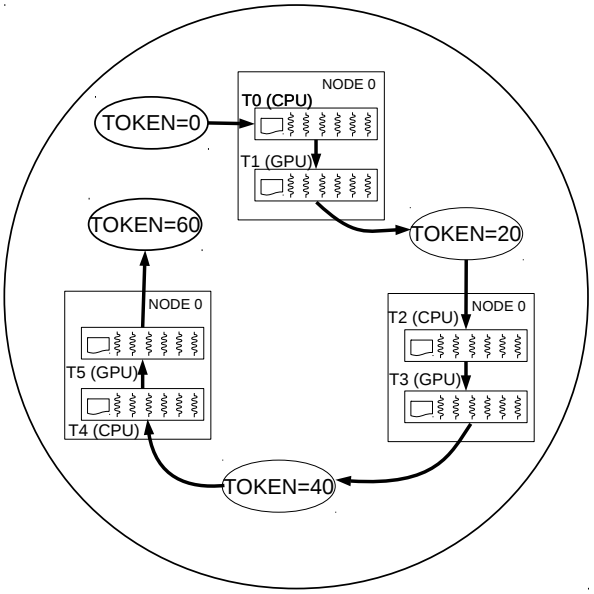


Figure A.2: Physical implementation of the task-based token ring application.

The code required to implement the token ring application is shown in Listing: A.1. In line 10 we get the number of tasks initialized in the system, line 12 sets the procedure for each task. In lines 14 to 16 each task allocates space for the token. In line 19 each task executes the procedure over the token, line 22 sets a synchronization point and line 23 performs the transfer of the token. In line 27 we synchronize the global execution to ensure that it has finished, and finally in lines 28 and 29 we recover and print the final value of the token .

The `Accum` procedure assigned to each task of the ring is depicted in Listing: A.2 and works as follows: In line 4 the tread checks if it is the first thread in the first task, if true it will initilize the token to zero, next in line 6 each thread atomically increments the value of the token.

Some of the remarkable points of this code are:

1. No device management code is required.
2. The scalability of this application is granted because adding more task to the system just requires a modification in the configuration file.

```
MPI_Init(&argc,&argv);
int taskId,
    results;
int token=0;
int numTasks;
int Dims = 1;
size_t globalDims[] = { numThreads };
size_t localDims[] = { numThreads };

err=XSCALA_GetNumTasks(&numTasks, MPI_COMM_WORLD);
int *taskDeps=malloc(sizeof(int));

for (taskId = 0; taskId < numTasks; taskId++) {
        err |= XSCALA_SetProcedure(MPI_COMM_WORLD,"tokenAdd.cl","Accum");
    err |= XSCALA_MallocTray(taskId, token, sizeof(int),MPI_COMM_WORLD );
}

for (taskId = 0; taskId < numTasks; taskId++) {
        err |= XSCALA_ExecTask(MPI_COMM_SELF, taskId, Dims, globalDims,
                    localDims, "%T ,%d", 0,taskId);
        *taskDeps = taskId;
        err |= XSCALA_WaitFor(1, taskDeps, MPI_COMM_WORLD );
        err |= XSCALA_SendRecv(taskId, token,
                        (taskId+1)%numTasks, token, 1,MPI_INT, MPI_COMM_WORLD );
}

err |= XSCALA_WaitAllTasks(MPI_COMM_WORLD );
err |= XSCALA_ReadTray(numTasks-1, 0, sizeof(int), &results, MPI_COMM_WORLD );
printf(" %d ", results);

MPI_Finalize();
}
```

Listing A.1: Token Ring Application

```
#pragma OPENCL EXTENSION cl_khr_local_int32_base_atomics : enable
__kernel void Accum(__global int* token, const int TaskId){

if (TaskId==0)
    token[0]=0;
atom_inc(&token[0]);
}
```

Listing A.2: Token Ring Task Procedure

Appendix B

Tiled Matrix Multiplication

Let A, B and C to represent three matrices with sizes $M \times K$, $K \times N$, and $M \times N$ respectively, and lets define a tile as a new submatrix embedded in A, B and C as is depicted in the Figure: B.1.

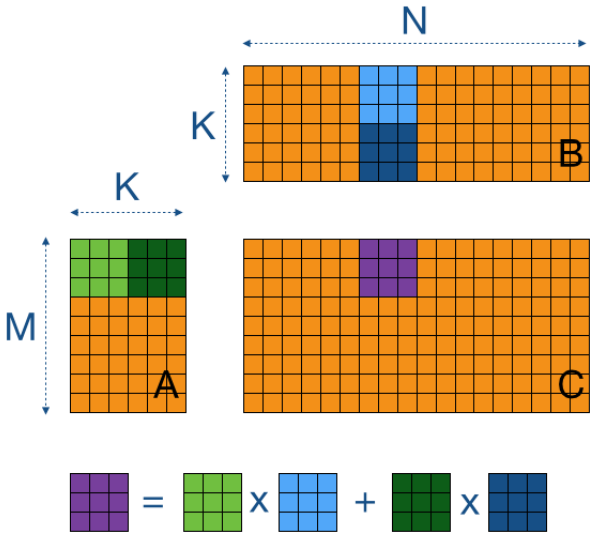


Figure B.1: Tiled matrix multiplication

In the naive schoolbook matrix multiplication method we launch as many threads as entries in the C matrix, and each thread computes one entry thus requiring a total of $2 \times K \times N \times M$ global memory loads and $M \times N$ memory stores to perform $2 \times M \times K \times N$ multiplications and additions yielding a granularity $O(1)\frac{ops}{access}$. In the tiling approach each thread can perform $2 \times TS$ operations with 2 global memory accesses yielding a granularity $O(TS)\frac{ops}{access}$ with TS representing the size of the tile enhancing the performance of the application.

The source code for the implementation of tiled matrix multiplication multiplication is shown in the Listing:B.1 and works as follows: In lines 17 to 26 each thread in the task must copy an element from the tiles in A and B into shared memory space and then in lines 28 and 29 each thread performs the inner product of its row in A by its column in B as is depicted in the Figure: B.2.

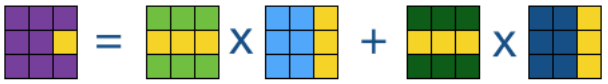


Figure B.2: Work per thread using tiles of size 3×3

```
1
2  #define TILE_WIDTH 16
3
4  __kernel void mtxMultShared(__global float * A,__global float * B,__global float * C
       ,const int numRows,const int numColumns) {
5
6          __local float ds_M[TILE_WIDTH][TILE_WIDTH];
7          __local float ds_N[TILE_WIDTH][TILE_WIDTH];
8
9          int tx = get_local_id(0);
10         int ty = get_local_id(1);
11
12         int Row = get_global_id(1);
13         int Col = get_global_id(0);
14
15         float Accum = 0;
16
17         for (int m = 0; m < (numColumns - 1) / TILE_WIDTH + 1; ++m) {
18                 if (Row < numRows && m * TILE_WIDTH + tx < numColumns)
19                         ds_M[ty][tx] = A[Row * numColumns + m * TILE_WIDTH + tx];
20                 else
21                         ds_M[ty][tx] = 0;
22                 if (Col < numColumns && m * TILE_WIDTH + ty < numRows)
23                         ds_N[ty][tx] = B[(m * TILE_WIDTH + ty) * numColumns + Col];
24                 else
25                         ds_N[ty][tx] = 0;
26
27                  barrier(CLK_LOCAL_MEM_FENCE);
28                 for (int k = 0; k < TILE_WIDTH; ++k)
29                         Accum += ds_M[ty][k] * ds_N[k][tx];
30                 barrier(CLK_LOCAL_MEM_FENCE);
31         }
32         if (Row < numRows && Col < numColumns)
33                 C[Row * numColumns + Col] += Accum;
34 }
```

Listing B.1: Tiled matrix multiplication procedure.

Bibliography

Adam, T. L., Chandy, K. M., and Dickson, J. R. (1974). A comparison of list schedules for parallel processing systems. *Commun. ACM*, 17(12):685–690.

Aji, A. M., Dinan, J., Buntinas, D., Balaji, P., Feng, W.-c., Bisset, K. R., and Thakur, R. (2012). Mpi-acc: An integrated and extensible approach to data movement in accelerator-based systems. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 647–654. IEEE.

Alves, A., Rufino, J., Pina, A., and Santos, L. (2013). clopencl - supporting distributed heterogeneous computing in hpc clusters. In *Euro-Par 2012: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 112–122. Springer Berlin Heidelberg.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM.

Aoki, R., Oikawa, S., Tsuchiyama, R., and Nakamura, T. (2010). Hybrid opencl: Connecting different opencl implementations over network. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2729–2735.

Armstrong, R., Hensgen, D., and Kidd, T. (1998). The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Heterogeneous Computing Workshop, 1998. (HCW 98) Proceedings. 1998 Seventh*, pages 79–87.

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198.

Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., and Zhang, G. (2008). A proposal for task parallelism in openmp. In *A Practical Programming Model for the Multi-Core Era*, pages 1–12. Springer.

Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. (2009). The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418.

Balaji, P. (2015). *Programming Models for Parallel Computing*. Scientific and Engineering Computation. MIT Press.

Barak, A., Ben-Nun, T., Levy, E., and Shiloh, A. (2010). A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7.

Bernstein, P. A. (1996). Middleware: A model for distributed system services. *Commun. ACM*, 39(2):86–98.

Blazewicz, J., Lenstra, J. K., and Kan, A. R. (1983). Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24.

Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., et al. (2001). A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837.

Brucker, P. (2013). *Scheduling Algorithms*. Springer Berlin Heidelberg.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

Cederman, D. and Tsigas, P. (2011). Dynamic load balancing using work-stealing. *GPU Computing Gems Jade Edition*, pages 485–499.

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition.

Courtois, P. J., Heymans, F., and Parnas, D. L. (1971). Concurrent control with “readers” and “writers”. *Commun. ACM*, 14(10):667–668.

Curry, E. (2004). Message-Oriented Middleware. In Mahmoud, Q. H., editor, *Middleware for Communications*, chapter 1, pages 1–28. John Wiley and Sons, Chichester, England.

Cybenko, G. (1989). Dynamic load balancing for distributed memory multiprocessors. *Journal of parallel and distributed computing*, 7(2):279–301.

Dongarra, J. and Lastovetsky, A. L. (2009). *High Performance Heterogeneous Computing*. Wiley-Interscience, New York, NY, USA.

El-Ghazawi, T., Carlson, W., Sterling, T., and Yelick, K. (2005a). *UPC: Distributed Shared Memory Programming*. Wiley Series on Parallel and Distributed Computing. Wiley.

El-Ghazawi, T., Carlson, W., Sterling, T., and Yelick, K. (2005b). *UPC: Distributed Shared Memory Programming*. Wiley Series on Parallel and Distributed Computing. Wiley.

El-Rewini, H. and Abd-El-Barr, M. (2005). *Advanced Computer Architecture and Parallel Processing*. Wiley Series on Parallel and Distributed Computing. Wiley.

Elangovan, V. K., Badia, R. M., and Ayguadé, E. (2014). Scalability and parallel execution of ompss-opencl tasks on heterogeneous cpu-gpu environment. In *Supercomputing*, pages 141–155. Springer.

Fernández, A., Beltran, V., Martorell, X., Badia, R. M., Ayguadé, E., and Labarta, J. (2014). Task-based programming with ompss and its application. In *Euro-Par 2014: Parallel Processing Workshops*, pages 601–612. Springer.

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

Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. (2004b). Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer.

Gary, M. R. and Johnson, D. S. (1979). Computers and intractability: A guide to the theory of np-completeness.

Goglin, B. (2008). Design and implementation of open-mx: High-performance message passing over generic ethernet hardware. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7.

Graham, R., Kan, A., and Institute, E. U. R. E. (1979). *Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey*. Reprint series / Erasmus University Rotterdam. Erasmus University.

Grasso, I., Pellegrini, S., Cosenza, B., and Fahringer, T. (2014). A uniform approach for programming distributed heterogeneous computing systems. *Journal of parallel and distributed computing*, 74(12):3228–3239.

Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

Herlihy, M. and Shavit, N. (2012). *The Art of Multiprocessor Programming*. Morgan Kaufmann. Morgan Kaufmann.

Hu, T. C. (1961). Parallel sequencing and assembly line problems. *Operations research*, 9(6):841–848.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High-Performance Programming*. Elsevier Science.

Kale, L. V. and Zheng, G. (2009). Charm++ and ampi: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282.

Kegel, P., Steuwer, M., and Gorlatch, S. (2012). dopencl: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 174–186.

Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. (2012). Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA. ACM.

Kirk, D. and Hwu, W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach*. Elsevier Science.

Krakowiak, S. (2007). Middleware architecture with patterns and frameworks.

Kruatrachue, B. and Lewis, T. (1987). Duplication scheduling heuristics (dsh): A new precedence task scheduler for parallel processor systems. *Oregon State University, Corvallis, OR*.

Kshemkalyani, A. and Singhal, M. (2011). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.

Kwok, Y.-K. and Ahmad, I. (1996). Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521.

Kwok, Y.-K. and Ahmad, I. (1999). Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422.

Lattner, C. (2008). Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2.

Lawlor, O. (2009). Message passing for gpgpu clusters: Cudampi. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323.

Leijen, D., Schulte, W., and Burckhardt, S. (2009). The design of a task parallel library. *Acm Sigplan Notices*, 44(10):227–242.

Maheswaran, M., Ali, S., Siegal, H. J., Hensgen, D., and Freund, R. (1999). Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 30–44.

Marowka, A. (2010). Chapter 2 - pitfalls and issues of manycore programming. volume 79 of *Advances in Computers*, pages 71 – 117. Elsevier.

McCool, M., Robison, A., and Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann. Morgan Kaufmann.

Mellor-Crummey, J. M. and Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with cuda. *Queue*, 6(2):40–53.

Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1994). The information bus: an architecture for extensible distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 58–68. ACM.

Quinn, M. (2004). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill.

Raynal, M. (2012). *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated.

Reinders, J. (2007). *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly.

Rinard, M. C., Scales, D. J., and Lam, M. S. (1993). Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38.

Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* John Wiley & Sons, Inc., New York, NY, USA, 2nd edition.

Scogland, T. R., Feng, W.-c., Rountree, B., and de Supinski, B. R. (2014). Coretsar: Adaptive worksharing for heterogeneous systems. In *Supercomputing*, pages 172–186. Springer.

Shirahata, K., Sato, H., and Matsuoka, S. (2010). Hybrid map task scheduling for gpu-based heterogeneous clusters. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 733–740.

Sih, G. C. and Lee, E. A. (1993). A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE transactions on Parallel and Distributed systems*, 4(2):175–187.

Song, F. and Dongarra, J. (2012). A scalable framework for heterogeneous gpu-based clusters. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 91–100, New York, NY, USA. ACM.

Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356.

Sun, E., Schaa, D., Bagley, R., Rubin, N., and Kaeli, D. (2012). Enabling task-level scheduling on heterogeneous platforms. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 84–93, New York, NY, USA. ACM.

Takizawa, H., Sugawara, M., Hirasawa, S., Gelado, I., Kobayashi, H., and Hwu, W.-M. (2013). clmpi: An opencl extension for interoperation with the message passing interface. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1138–1148.

Tanenbaum, A. and van Steen, M. (2007). *Distributed Systems: Principles and Paradigms.* Pearson Prentice Hall.

Topcuouglu, H., Hariri, S., and Wu, M.-y. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274.

van der Aalst, W., ter Hofstede, A., Kiepuszewski, B., and Barros, A. (2003). Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51.

Vasudevan, R., Vadhiyar, S. S., and Kalé, L. V. (2013). G-charm: An adaptive runtime system for message-driven parallel applications on hybrid systems. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 349–358, New York, NY, USA. ACM.

Wilson, R. J. (1996). *Introduction to Graph Theory*. John Wiley & Sons, Inc., New York, NY, USA.

Wu, M.-Y. and Gajski, D. D. (1990). Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, 1(3):330–343.

Yang, T. and Gerasoulis, A. (1994). Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967.