RUNNING PARALLEL APPLICATIONS ON A HETEROGENEOUS ENVIRONMENT WITH ACCESSIBLE DEVELOPMENT PRACTICES AND AUTOMATIC SCALABILITY

by

Jeremy Francisco Villalobos

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Information Technology

Charlotte

2011

Approved by:

_____
Dr. Barry Wilkinson

_____
Dr. Jing Xiao

_____
Dr. Yu Wang

_____
Dr. Monica Johar

ABSTRACT

JEREMY FRANCISCO VILLALOBOS. Running parallel applications on a heterogeneous environment with accessible development practices and automatic scalability. (Under direction of DR. ANTHONY BARRY WILKINSON)

Grid computing makes it possible to gather large quantities of resources to work on a problem. In order to exploit this potential, a framework that presents the resources to the user programmer in a form that maintains productivity is necessary. The framework must not only provide accessible development, but it must make efficient use of the resources. The Seeds framework is proposed. It uses the current Grid and distributed computing middleware to provide a parallel programming environment to a wider community of programmers. The framework was used to investigate the feasibility of scaling skeleton/pattern parallel programming into Grid computing. The research accomplished two goals: it made parallel programming on the Grid more accessible to domain-specific programmers, and it made parallel programs scale on a heterogeneous resource environment. Programming is made easier to the programmer by using skeleton and pattern-based programming approaches that effectively isolate the program from the environment. To extend the pattern approach, the pattern adder operator is proposed, implemented and tested. The results show the pattern operator can reduce the number of lines of code when compared with an MPJ-Express implementation for a stencil algorithm while having an overhead of at most ten microseconds per iteration. The research in scalability involved adapting existing load-balancing techniques to skeletons and patterns requiring little additional configuration on the part of the programmer. The hierarchical dependency concept is proposed as well, which uses a streamed data flow programming

model. The concept introduces data flow computation hibernation and dependencies that can split to accommodate additional processors. The results from implementing skeleton/patterns on hierarchical dependencies show an 18.23% increase in code is necessary to enable automatic scalability. The concept can increase speedup depending on the algorithm and grain size.

ACKNOWLEDGEMENT

Thanks to Dr. Wilkinson for his advice and discussion on many topics. Thanks to Monika Lakha and Satya Venkata for working on aspects of the framework. Thanks to Saurav Bhattarai for validating the framework by implementing a bio-informatics algorithm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API: Application Programming Interface

CO_2P_3S: Correct Object-Oriented Pattern-based Parallel Programming System

CUDA: Compute Unified Device Architecture

DSM: Distributed Shared Memory

GPT: Grid Packaging Toolkit.

HPC: High Performance Computing

ID: Identification

JNI: Java Native Interface

JVM: Java Virtual Machine

LAN: Local Area Network

MPI: Message Passing Interface

NAT: Network Address Translation

NIC: Network Interface Card

NOW: Network Of Workstations

OOP: Object Oriented Programming

PDE: Partial Derivative Equation

PSE: Problem Solving Environment

RMI: Remote Method Invocation

SP: Skeletons and Patterns

TBB: Thread Building Blocks

TCP: Transmission Control Protocol

UML: Universal Modeling Language

UPNP: Universal Plug N' Play

WAN: Wide Area Network

XML: eXtensible Markup Language

CHAPTER 1: INTRODUCTION

Grid computing is the concept where multiple computing and sensor resources are shared among institutions in what are called virtual organizations (VO's) [1]. A VO is able to share data and computational resources through Grid tools that provide authentication, and access services. The promise of Grid computing is not only about computational power, but about sharing technological equipment such as telescopes, and particle accelerators; the Grid allows the scientists and users from different fields to communicate and work together on a problem independent of their geographical position.

Grid computing is also used to combine multiple Grid resources to work on a single problem. We will call a single geographical location in the VO a Grid node (GN). The Cactus project has used multiple Grid resources to work on physics problems, which in the future will include colliding black holes [2]. These examples require close coordination among the institutions, which in effect is equivalent to having a hyper-institution. Grids that coordinate their actions and software closely are called **federated Grids** because there is a central entity that directs how the institution's resources should be used. Projects such as Cactus [3] and TeraGrid [4] have shown that it is possible to run parallel applications among multiple Grid nodes that are geographically distributed. These projects use state-of-the-art fiber optics to communicate and supercomputers as the Grid resources. These types of high performance Grids are called **Lambda Grids**. They can

produce PetaFLOPS or 10^{15} FLOating Point operation per Second. A less regulated vision of the Grid has been termed as the **Grid economy** [5] where the computational resources are not coordinated by a central authority, but instead, are coordinated by a market where the resources can be rented at a fair market value[5]. The institutions in this scheme would decide the software they use based on their own needs, and the needs that provide a return from the market.

The Grid economy remains a concept, but federated Grids are in existence today. The use of federated Grids have provided multiple solutions to the problem of running parallel applications on the Grid. But the tools used by the TeraGrid and Cactus today are not enough to manage a real Grid. The resources used by these projects were more-or-less known quantities such that the programmers were able to code for the platform. The description of a Grid by Foster et al. in "Anatomy of the Grid" suggests a Grid is made up of multiple resources, some of which may not be stable, and the network connections may be shared with other traffic [1]. A program created with conventional tools will likely under-perform in such a Grid environment, or not be able to deploy in the first place. Moreover, if a new group of scientists has a project for a different Grid environment, they too would need to learn the different variables of performance and connectivity all over – in essence, redo the research and development that the mentioned projects have done. Projecting parallel development for Grid applications onto the current concept of the Grid would require programmers and scientist to become Grid computing experts. This type of steep learning curve is, I believe, the roadblock that has prevented the adoption of Grid computing as a parallel application platform over the last decade.

The roadblock also includes the connectivity between nodes, which can be prohibitively slow for some types of applications. Currently we consider work pool programs to be feasible for a Grid environment, and an N-body type problem to be not feasible. Somewhat coupled algorithms should be feasible on the Grid, but in order for that to happen, the algorithm needs to adapt to the IO characteristics. The coupled parallel program must find a set of processors that have similar latency and bandwidth among themselves before they start running on them, or adjust while it is running. The process of configuring and load balancing such an algorithm cannot be left up to the programmer, since that would discourage programming with this environment.

Grid computing has moved toward a service oriented approach. This, in part, is because of the difficulty of using multiple grid resources for a single problem. The input/output (IO) tends to get exponentially slower going from the internal CPU speeds to the network and storage devices. Figure 1.1 shows the complexity of today's heterogeneous environment. Today the communication between cores inside of a processor tends to be in the tens of Gigabytes per second (GB/s), the communication between the processor and the bus is less than 10 GB/s, and network and hard drive communication goes down to less than 100 MB/s for most hardware. Although these fluctuations in the bandwidth and latency do not make parallel programs infeasible for the Grid, they do add complexity that could turn into inefficiency if not resolved. Developing for such an environment adds multiple variants to complicate development and deployment.

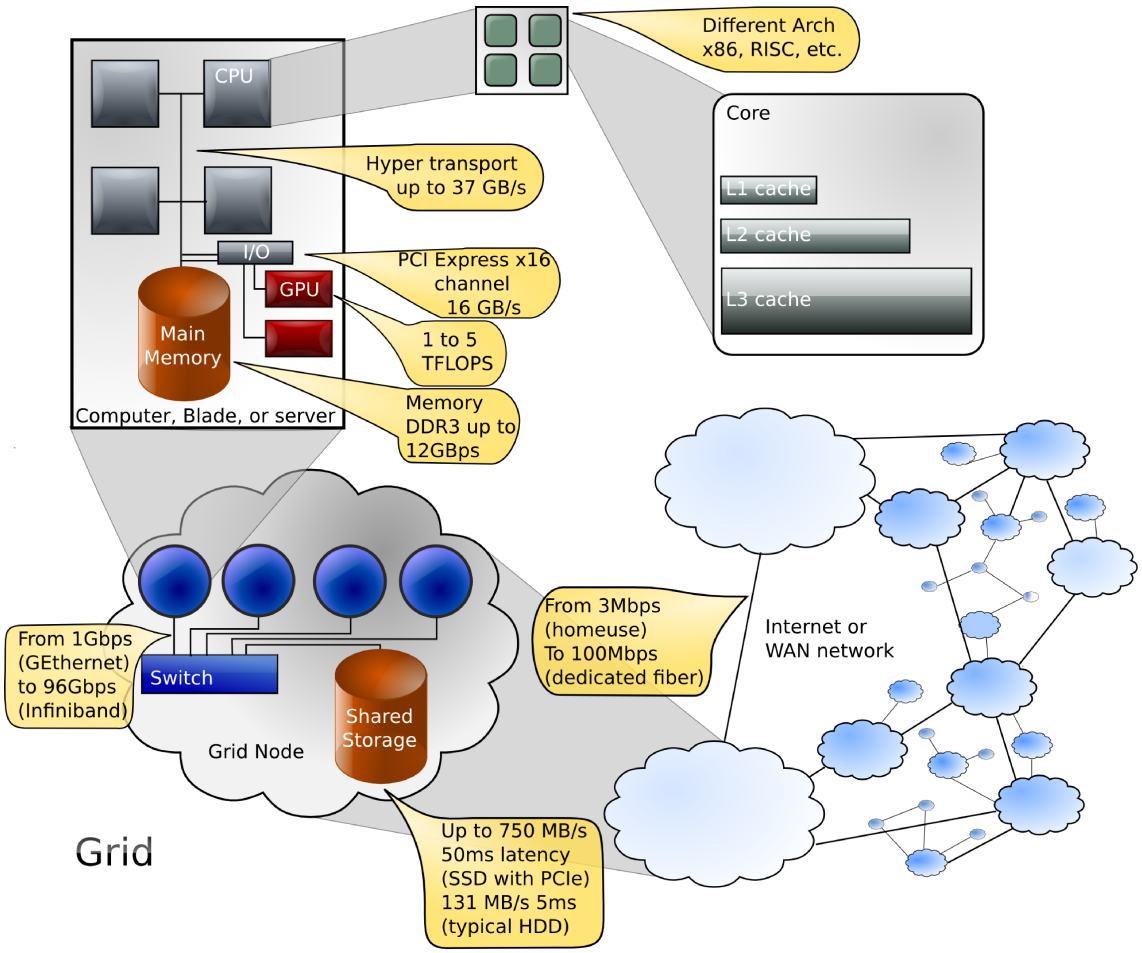


FIGURE 1.1: A high level view of the heterogeneous environment.

Some of the non-functional requirements that are part of a heterogeneous environment include:

- Network characteristics
  - Connectivity
  - Bandwidth
  - Latency
- Processor characteristics
  - Architecture
  - Speed

- Acceleration
- Security
  - Authentication
  - Secure Communication
- Fault tolerance
- Miscellaneous
  - Main memory capacity
  - Hard drive speed

These requirements will be covered in more detail in the Section 1.2.3, where the concepts are explained along with the known literature on addressing the challenges pertinent to each issue.

Chapter 1 is organized as follows. Section 1.1 briefly reviews the skeleton/pattern programming approach. Section 1.2 reviews the literature on the topics touched by the research. Section 1.3 describes contribution from the dissertation. For reference, Section 1.4 identifies topics in Grid computing that are outside the scope of the dissertation. Section 1.5 explains the expected impact of the work on Computer Science and other fields that are influenced by Computer Science.

1.1 Skeletons and Patterns

Although the challenges posed by input/output are solved with existing load-balancing techniques or through trivial inventions of new ones, the parallel application must also be developed through a programming style that is easy to learn and use. The development of Grid applications may be expanded to different fields of research and industry if the development styles shield the developer from the complexities that have to be addressed in

Grid parallel applications. There have been different efforts done to abstract communication procedures and the environment's complexity; some of these efforts are reviewed in Sections 1.2. The area of skeleton/pattern programming as a way to abstract the environment is the area of research for this dissertation.

The start and end of the patterns are the same as the skeleton, that is, they start with some mapping of the initial data to the processing nodes, and they end by converging the processed data into a sink node. Figure 1.2 shows the life cycle for both skeletons and patterns. The model we use assumes a single source point and a single sink node. A program starts by assigning tasks to each of the worker nodes, then the sink begins streaming input data into the nodes. The nodes compute on the data and emit an output stream. The final answer is collected at the sink.

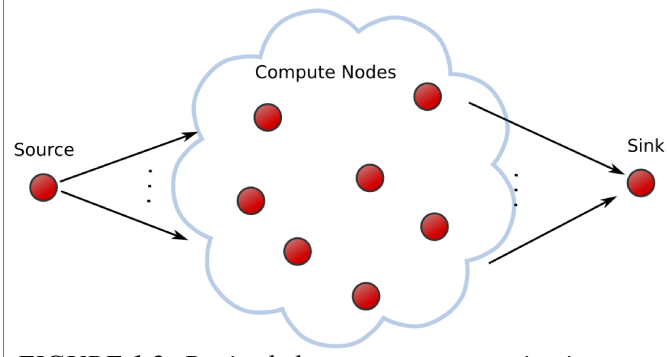


FIGURE 1.2: Basic skeleton pattern organization.

Skeletons are structured programs that resemble tree structures and are applied to specific known types of recurring parallel problems. Skeletons can be modeled with directed, acyclic graphs, and their implementations are data parallel. Skeletons are analyzed at the theoretical level using functional programs like Haskel and $\lambda - caculus$ [6]. The algorithms create a flow of data that streams from one stage to the other until the necessary algorithms have been performed to the data. Figure 1.3 shows a graphical representation

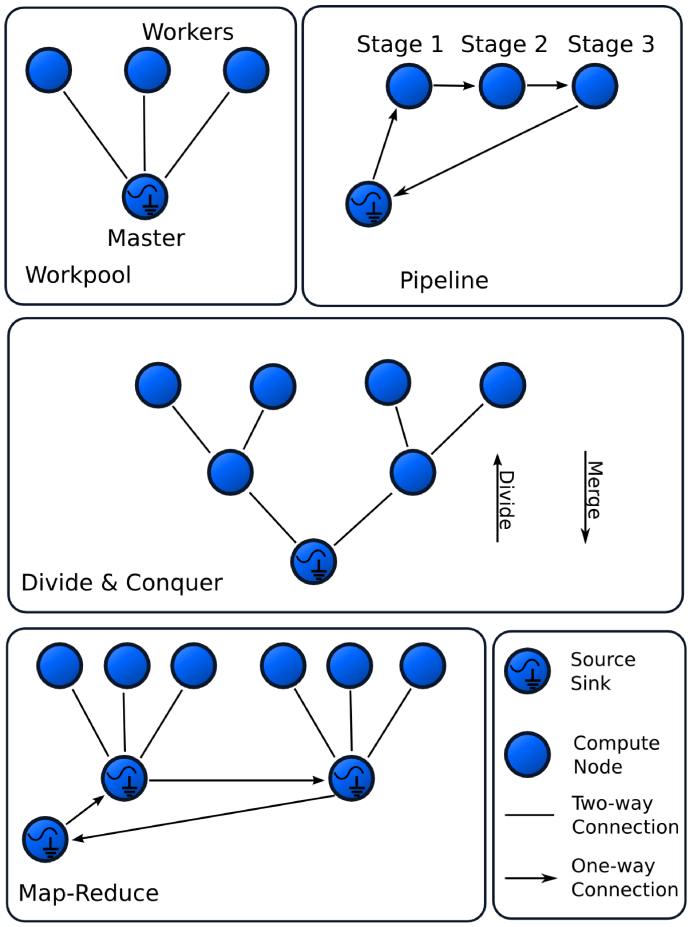


FIGURE 1.3: Examples of skeletons for parallel programs.

of the most common skeletons.

From left to right and from top to bottom, the skeletons shown in the figure are:

- Workpool (farm, parameter sweep): The data is segmented into multiple independent pieces, the master sends these pieces to each worker until there are no more pieces to process. The master receives the processed pieces and returns that data to the user programmer. Chapter 2 presents the implementation done on our framework.
- Pipeline: The data is segmented into multiple independent pieces. The work is performed on each piece in several stages. The next stage requires for the data to

have been processed by the preceding stage. When the data goes through the last stage, the data is returned to the user programmer. Chapter 4 presents the implementation done for our framework.

- Divide & Conquer: The data is divided into some amount of pieces, usually two, until the data comes down to a size where it can be processed with reduce computational time. After computing the data, the cycle is repeated backwards to merge it into the final result.
- Map-Reduce [7]: The data is divided into pieces and sent to each workers as is done in a workpool. The workers are not expected to finish the job, but to advance the job to an intermediate state. Then, the intermediate data is allocated depending on its state to another worker which "reduces" the answer by integrating the answers from multiple intermediate data pieces. The final answer is a list of categories created during the map phase with values that where summed up by the reduce phase.

Multiple projects have implemented the skeletons in different languages and different framework for different environments [8],[9],[10],[11]. Map-reduce[7] by Google™, can be considered a skeleton, and much research has been done to find out how to adapt more algorithms to it [6].

Patterns are very similar to skeletons. Like skeletons, patterns are created based on frequently recurring parallel programming design structures. However, patterns cannot be modeled using functional programming because the nodes in the pattern have a two-way connection, and some data state has to be kept during the computation and commu-

nication. Patterns can be modeled using cyclic, undirected graphs, and like skeletons, patterns are data-parallel. Figure 1.4 shows graphic representations of the pattern in the following non-exhaustive list:

- Stencil (Also known as geometric decomposition or synchronous programs): It is used to processed numerical approximation to PDE such as the heat equation. The pattern distributes initial states to a number of processors, the processors loop around a function for some arbitrary number of cycles. When the cycles are done, the data is sent back to the master. Chapters 3 and 4 present two different implementations of this pattern done on the Seeds framework.
- N-body (all-to-all): The algorithm is used for processes that require synchronization with all its neighbors. A known application is the naive implementation for the N-body problem. The procedure is the same as the stencil, but synchronizes with all the nodes, not just the close neighbors. Chapter 3 presents an implementation of the all-to-all pattern done in our framework.

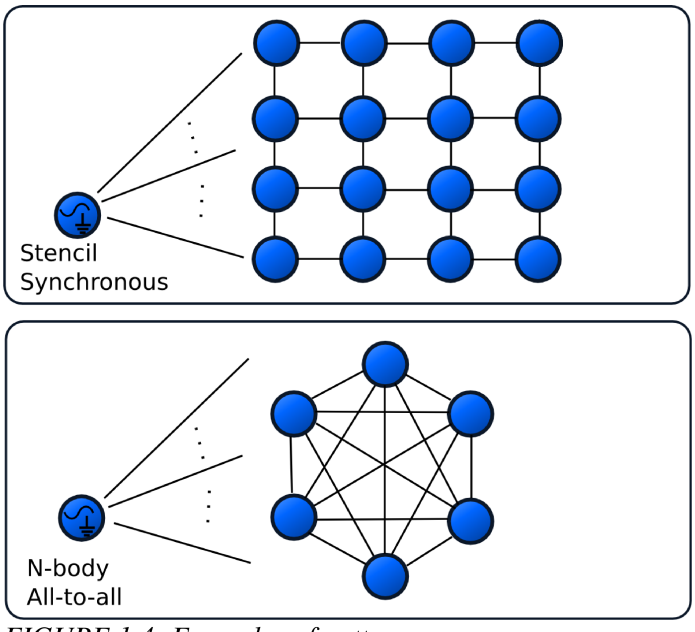


FIGURE 1.4: Examples of patterns.

There are three advantages to using skeleton/patterns over the industry standards such as MPI [12], OpenMP [13], and explicit thread libraries:

- Reduce deadlock and race conditions: Skeleton/patterns provide implicit parallelization to the user programmer. This is done by giving the domain-specific programmer an interface. When the domain-specific programmer gives the implementation to a framework, the framework will run the interface while taking care of the race conditions and deadlock.
- Reduction in code and development time: Macdonald et al. showed that coding with patterns requires less coding, and is simpler in comparison to MPI [14]. Aldinucci et al. also have shown frameworks such as Lithium and Muskel that use object-oriented Java to provide the benefits of skeletons to the user programmer [9]. Object-oriented languages have created the abstraction that is necessary for the concepts behind SPs to be provided in a form that is simpler to understand. Previously, projects such as eSkel [15], Sketo [10], and DpnDP [16] provided skeletons using procedural C/C++ (which is not strongly typed therefore the data can be casted improperly) but they create an environment where mistakes are hard to find [16].
- Abstracts the parallelization: The third advantage for skeleton/patterns is due to the increased need to abstract the parallelization away from the computational resources as is needed in cloud and Grid computing. The abstraction is needed mainly in Grid computing because the environment is made up of multiple architectures and network topologies. Ideally, service providers want an application to

run on this environment while minimizing the amount of knowledge the user programmer needs to code it. In the case of cloud computing, the environment tends to be more homogeneous and controlled by a single entity, but the service provider also wants to provide the computation resources in a way that they can optimize the use of the hardware. The hardware optimization leads to servicing more customers. Skeleton/patterns are a pertinent option to abstract the use of the resources because they allow the programmer to code the problem using the provided API, and they give the Grid/Cloud maintainers space to manage the non-functional requirements. The API and interfaces, in effect, create an extra layer between the hardware and the user programmer.

Skeletons/patterns were ignored in favor of message passing and share memory techniques in the industry for many years since the concept was first introduced in academia. More recent developments have introduced the workpool skeleton to the industry by Nvidia and Intel, with the projects Compute Unified Device Architecture (CUDA)[17], and Threaded Building Blocks (TBB)[18] respectively, but at the moment they are relegate to only one computer device. The next section presents a thorough but not complete survey on skeletons, patterns and their implementations of multiple frameworks.

1.2 Related Work

The problem of running parallel applications on the Grid has been addressed in multiple forms since the creation of the Grid concept and some approaches date back to early distributed computing and Networks Of Workstations (NOWs). The projects we will cover mention both the attempts at simplifying development and increasing scalability. This section is divided into three subsections that focus on the importance of particular aspects

of frameworks, their ability to adapt to the environment, and their ability to facilitate development by the user programmer. The sections are programmability, memory management and scalability.

1.2.1 Programmability

We refer to programmability as a measure of how easy it is to code the parallel program on that framework or style of programming. Measuring programmability is done by combining subjective and objective measurements. An objective measurement is the use of lines of code (LOCs). The less LOCs it takes to code a parallel version of a given program, the easier the programming style is considered. Another approach is to conduct surveys with programmers, and ask them what they think about the programming style they have been exposed to [19]. Over the years, there have been multiple approaches to programming parallel applications, with a select few used by industry today. This subsection divides programming styles by the following categories: parallel programming languages, programming languages, language properties, language extensions, and automatic code generation.

Parallel Programming Languages: There are multiple levels from which researchers have attempted to simplify development. At the lowest level, there is the modification or invention of new programming languages that better transfer the use of parallel directives. There are no mainstream languages in use today that can be used as an example of a programming language with dedicated keywords to denote parallelization. Most of the popular languages today, C, C++, and Java do not have these keywords. Instead, they rely on other methods. These three programming languages are the focus for this section.

Programming Languages: In order of programmability from worst to best, one can arrange the main languages just mention as: C, C++, Java. However there is a trade-off. In order of performance, the same languages going from worst to best are: Java, C++, C. Generally, it is safe to assume that a language with a simple syntax like C will always outperform a language like C++ or Java because the user gets access to the lower levels of the hardware. Bull et al. shows that Java can be 7% slower than C, and "the best Java execution time is within 20% of the best Fortran" [20]. This can be interpreted to mean that we are in perpetual balancing cycle between ease of programmability and performance.

Language Properties: Parallel programming can be incorporated into a language without the need to directly modify the language. This can be done by adding libraries and/or creating frameworks. Object-oriented programming languages such as Java and C++ make it easier to develop frameworks by abstracting tasks and modularizing code. Frameworks can create isolation from the networking interfaces and network protocols, simplifying development. For example, the MPI send directive abstracts the processors by assigning them a rank. The user does not need to know the IP or port for the particular machine running a process. Instead, the user mentions the processor's rank. The framework then fills in the necessary information to route the message. Similarly, frameworks created with Java can increase the abstraction by providing the user with skeletons and patterns. By using those, the user is able to accomplish algorithms that can be parallelized without having to micro-manage the parallel computation. There are several projects that have implemented skeleton frameworks [21], [15], [9], [10], and pattern frameworks

[14],[16],[11], with C, C++ and Java. Aldinucci et al. have researched many aspects on skeletons and patterns like managing non-functional requirements, the creation of a labeled transition system language to convert skeletons in functional notation to a representation that better shows the parallelization [22]. Skeletons focus more on small grain size parallel algorithms, and patterns focus more on coarse grain size and stateful algorithms. The skeleton and pattern definitions are sometimes used interchangeably as in the PASM project [11]. The main disadvantage of using skeletons/patterns is that there may not be a fundamental set of patterns that can be used to create all other patterns. This means that the list of patterns from which a programmer has to choose may be large, which may discourage a programmer from using patterns. On the other hand, if the library is small, the programmer may have to create a pattern using lower level APIs; at which point, the development may look similar to developing the parallel application with a lower-level API.

Language Extensions: Some approaches conclude that the programming language is not enough to reduce the effort to create a program, and other languages and tool sets are added to a language to improve development efficiency. The two discussed here are the use of preprocessing languages, and the use of aspect-oriented programming (AOP). A language requires the use of a preprocessing language to create programs that can run on multiple platforms. In effect, the developers can superimpose the preprocessing code that can create the code for the final program during compile time. OpenMP makes use of a preprocessing language to insert valid C code to accomplish parallel instructions just before the program is compiled. OpenMP works primarily on Fortran and C languages. MPI

does not use preprocessing languages for parallelization. Cole's eSkel provides keyword aids that can be turned into C code before compiling the code [15].

Aspect-oriented programming (AOP) is proposed as an extension to Object-Oriented Programming (OOP). It can also be seen as a development aid. During program development, parallel or otherwise, programmers come across pieces of code that are needed constantly in multiple places throughout the program. The literature calls a piece of code that manages a particular aspect, a *concern*. Concerns that are mentioned on multiple places in the code are called *cross-cutting* concerns. Using AOP, the user is able to call specific pieces of code before a function, and after a function. AOP has specific language rules design to give a programmer control over the code he would not have using OOP. Some projects have identified parallel programming as a cross-cutting concern and have provided frameworks to explain how the approach works [23], [24].

Automatic Code Generation**:** There are some instances where development tools such as an integrated development environment (IDE) can create portions of code based on information the programmer provides into a wizard. This practice is particularly popular with the development of graphic user interfaces (GUIs). The IDE presents the user with a graphical representation of the program's window, the user modifies the window, and the changes get ported into code. Other assistance includes automatically adding methods the programmer must add after implementing an interface . MacDonald et al. also applies this idea to the creation of parallel program for a cluster of computers [14]. The user picks a parallel programming pattern from a list of provided patterns. The IDE creates the code automatically and then presents the programmer with a window that graphically rep-

resents the pattern. The user is able to customize the pattern to solve the problem at hand. Although the use of a GUI and automatic code generation facilitates development, there are a few drawbacks to the strategy. The use of a GUI to edit the code requires the introduction of a new language, perhaps an XML schema to reduce the GUI representation to a text file. This would be necessary so the programmer has choice over the IDE he would use to create a program. At the same time, this would complicate development for a programmer that does not want to use a GUI to create a program. CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System) also converts the pattern from a language-neutral form to code in some language selected by the user, which includes C++ and Java. This approach has the advantage of language independence as well as platform independence. One drawback is that the code is likely to be never optimal for any language, and to achieve the expected performance, the programmer may have to modify the automatically generated code.

Table 1.1 shows the projects reviewed for this literature search, organized based on their main programming style. They were further clustered based on the parallel programming style they used such as MPI, OpenMP, patterns, and skeletons.

TABLE 1.1: Projects organized according to their developmental qualities. The rows are categorized by parallel programming style, and the columns are categorized by the programming aid used to deliver that style.

	Preprocessing Language	***GUI tools***	***Libraries***	***Framework***
MPI			MPICH-G2, GridMPI, MPICH-V2, P2PMPI, P3, Global Arrays Toolkit, MPJ/Ibis	
OpenMP	OpenMP			
Patterns		CO_2P_3S		
			DPnDP, Grid.It, PASM	
Skeletons			Lithium, SkeTo	
	eSkels		eSkels	

1.2.2 Memory Management

Memory management issues can be placed in the middle ground between programmability and scalability. This is because any memory management decision for parallel programming affects both of these fields. Distributed memory solutions typically make it harder for programmers to create a parallel program, and extending shared memory development practices to a cluster computing or the Grid reduces performance by increasing overhead. This section reviews the shared memory concept, the distributed memory concept, and it explains the literature on a concept called Global Arrays [25]. The review looks at both the performance and the development aspects in memory management. Figure 1.5 shows the memory management styles reviewed and the techniques that have been created to take advantage of them.

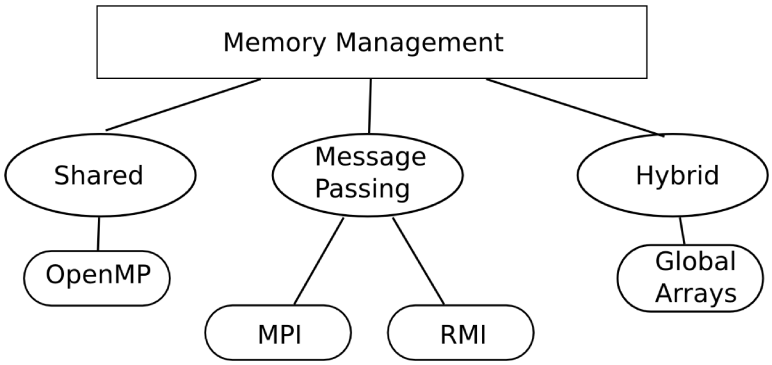


FIGURE 1.5: Types of memory management and existing standards or projects that use them.

Shared Memory: In a shared memory system, the main memory can be used by any of the processors. On the performance side, this means that multiple processors can work on a piece of data at practically the same time. This is in contrast to distributed memory where the synchronization has to happen by sending messages back and forth. Shared memory is also faster because the main memory is close to the processors. On the development side, shared memory is favored by programmers because the data structure does not need to be moved, and other techniques like caching and ghost zones are not necessary. Shared memory does need techniques to organize memory access so that the processors do not override each other's work. OpenMP provides data access management techniques that are hidden from the user by the preprocessing language. Java provides thread-based shared memory, and multiple classes provide thread-safe access to shared data structures. The user has to be aware of the data management issue when using Java. C and C++ provide thread-based shared memory access, although not as well integrated as Java threads are.

The main drawback to shared memory management is that it does not scale outside a multi-core system. Attempts in the past have shown expanding shared memory develop-

ment practices to clusters computing brings with it overhead that puts it behind message passing [26], [27]. On the development side, there are no real drawbacks to shared memory, the user does need to coordinate access to shared data structures, but the technique is close to ideal.

Distributed Memory: Cluster computer systems and Grid virtual organizations are made up of multiple computers. The advantage of running a program on distributed memory is that the programmer can use the aggregated memory of the whole system, which can vastly improve performance for certain problems. The technique is also able to dedicate multiple processors to a single task. The most popular standard to program for distributed memory systems is message passing using MPI. Remote Method Invocation (RMI) is another message-passing standard [28]. RMI initially started as a library to allow OOP languages to request services from a server using remote objects. RMI has been extended for parallel programming in projects such as Lithium [9], although the programmer's exposure to the RMI core behind the frameworks vary according to the project. Lithium uses skeletons as well, so the programmer does not have direct interaction with RMI.

On the downside, message passing is one of the techniques that presents the the most challenges to programmers. A program is hard to implement and troubleshoot using message passing. This is mostly because the programmer has to coordinate sends and receives from different code sections of the program. There are a few drawbacks to message passing on the scalability side as well, although these are due in large part to the hardware environment where message passing is used. Because network communication

is slower than the internal components of a processor, the latency has to be taken into account when increasing the resources that will run a program. As a result, frequent network calls should be limited. The bandwidth is also orders of magnitude lower over the network compared to the internal speed of a CPU; the programmer, therefore, has to be aware of how much data is going back and forth through the communication link. Lastly, the programmer is also advised to interleave computations and communication so that the processors are not idle while information is being exchanged across the network, and vice versa. All of these tweaks increase the level of knowledge expected from a parallel programmer, which also reduces the amount of programmers that are willing to use the medium.

Global Arrays [25]: In summary for the last two techniques, shared memory is easy to program but does not scale, and distributed memory systems can scale to very big systems if the programmer is very versed with parallel programming, but distributed memory is hard to program for most programmers. As stated before, there has been some efforts to expand shared memory development to bigger distributed memory systems with a low success rate. Global Arrays is an attempt to reach a compromise between shared memory and distributed memory by having the data being shared as is done in shared memory, but the synchronization steps are consciously activated by the programmer as is done with distributed memory. The goal is to avoid the overhead incurred by previous attempts by making the programmer more aware of the data locality, and to simplify development for distributed memory. Global Arrays have been studied mainly in the High Performance Computing (HPC) field.

The Grid and Cloud environments are not made up of only shared memory systems, or distributed memory systems, but it are made up of a mixture of these systems. If one of the previous two techniques is used for Grid parallel programming, the systems that do not adapt to that technique will underperformed. For example, we could treat all systems as if they all have distributed memory; but would reduce the possible benefits that the shared memory system is providing. This suggests the need for memory management similar to Global Arrays. Although the compromise can successfully integrate shared and distributed memory, the technique by itself is not enough as a programming approach for the heterogeneous environment as a whole. There are other non-functional requirements the domain-specific programmer is exposed to if an existing Global Arrays framework were to be used in the Grid.

Table 1.2 shows the projects discussed for this literature search along with the technique for memory management used. Most of them use the distributed memory, message passing technique. The project shown for the use of Global Arrays was created specifically to test the Global Array idea.

TABLE 1.2: *Parallel programming frameworks organized according to their interaction with the hardware resources.*

Project	Programming language	Programming Style	Network Type	Network Connectivity	Self- Deployment	Intended Use
CO_2P_3S[14]	C,C++, Java	Patterns	Static	None	No	Cluster
P2PMPI[39]	Java	MPI	P2P	None	No	Cluster
MPICH-G2[32]	C, C++	MPI	Static	Gateway	No	Grid
MPICH-2[52]	Fortran,C,C++	MPI	Static	None	No	Cluster
MPICH-V2[42]	C, C++	MPI	Static	None	No	Cluster
P3[40]	Java	MPI	P2P	None	No	Cluster
GridMPI[37]	C,C++	MPI	Static	None	No	Grid
Lithium[9]	Java	Skeletons	Static	None	No	Cluster
Grid.it[21]	Java	Skeletons	Static	None	No	Grid
SkeTo[10]	C++	Skeletons	Static	None	No	Cluster
OpenMP[13]	C,C++	OpenMP	N/A	N/A	N/A	Workstation
eSkels[15]	C	Skeletons/MPI	Static	None	No	Cluster

TABLE 1.2: *Parallel programming frameworks organized according to their interaction with the hardware resources.*

Project	Programming language	Programming Style	Network Type	Network Connectivity	Self- Deployment	Intended Use
PASM[11]	C++	Skeleton/Patterns	Static	None	No	Cluster
DPnDP[16]	C++	Patterns	Static	None	No	Cluster
GlobalArrays[25]	Fortran,C,C++	Global Arrays/MPI	Static	None	No	Cluster
MPJ/Ibis[35]	Java	MPI (MPJ)	Static	SmartSockets	No	Grid

1.2.3 Scalability

Scalability refers to the performance of a program on a diverse set of resources. This section reviews the performance side of choosing a programming language, the need for a Grid application to adapt to the network, and the unavoidable need to include load balancing, latency reducing, bandwidth reducing, and fault tolerant techniques to take advantage of the Grid.

Programming Language Performance: Programming languages can be divided into three groups: compiled, interpreted, and a hybrid called Just-in-Time complied (JIT). Java is a JIT language. The main advantage of Java for Grid computing is that it provides a solution for heterogeneous environments. A compiled binary of a Java program can be run equally well on multiple architectures. On the down side, Java is slower than C++, it may also not be equipped with features to exploit processor-specific instructions or accelerators. The use of accelerating instructions is not automatic on C, or C++, but the programmer is given access to libraries that take advantage of them. The disadvantages associated with C, and C++ pertain to the ease of programmability as noted on Section 3.1 and the overhead required to adapt the languages to a heterogeneous environment, namely having to check that the libraries required by a program exist for a particular platform, and compiling for that particular platform. Table 1.3 shows a summary of the programming language qualities.

TABLE 1.3: Popular languages for parallel computing and distributed computing.

	Object Oriented	Management of Free Memory (Garbage Collection)	Explicit Pointer Use	JIT,Compiled
Fortran	NO	No	Yes	Compiled
C	NO	No	Yes	Compiled
C++	Yes	No	Yes	Compiled
Java	Yes	Yes	No	JIT

Network Connectivity: These complexities can be summarized by two main issues: the use of Network Address Translation units (NATs) and the use of firewalls. There are ways to cope with these although they are stand-alone projects not integrated into frameworks [29], [30], [31]. This section explains the complexities and provides the topologies used by the literature's projects. Some of the isolated solutions to the network problems are discussed.

NAT: is a system that allows for the use of “private” addresses within an organization while allowing the private nodes to access WAN servers. The interaction is not symmetric. If the WAN host starts a connection, it will not be able to reach the client computer inside the NAT because the ports opened during the interaction are opened by the router in reaction to the internal computer. There have been proposals to resolve this problem. One way is to manually configure a router to forward a specific port to a computer inside the private network, the Universal Plug N' Play (UPNP) standard also allows the programs to configure routers to forward ports. This solution may not be available in institutions where security is managed strictly. The most used solution among existing projects is the use of gateways [32], [33]. This is done by designating a node, or having a new ser-

vice that routes packets from inside the NAT to other computers in the WAN network. Alexander et al. mentions the use of a hand-shake protocol that will guarantee the client computer starts the conversation [31]. The concept is known as TCP splicing and relies on both computers agreeing to start a connection thereby opening ports from the inside; the computers organize themselves through another channel that was more easily accessible. SmartSockets[34] were made to provide connectivity to Ibis, a Java MPI project [35]. SmartSocket provide a gateway and TCP splicing techniques to connect MPI nodes.

Firewall: The firewall issues can be addressed by manually opening the port needed for a program. This may be easy if the administrator owns all the resources, but it can be difficult when the resources are not administrated by one entity. The other approach is to scan for open ports and used the available open ports that are found. None of the projects reviewed address the firewall issue or provide solutions for it. None of the projects mention the problem that firewalls bring to Grid applications, although, MPICH-G2 requires the Globus Toolkit[36], which provides extensive documentation on configuring firewalls, and troubleshooting problems brought on by firewalls.

Topology: All the projects in the literature search need to either create a virtual network topology, or use the existing network. The network topology used by the projects can be divided into two groups. The first one uses the existing physical network directly and can be referred to as static. The second type uses a network overlay. The static network is made of computers connected by routers and Ethernet hubs. They represent the actual topology created in the physical world. They are also made up of abstractions like the NAT and the firewall mentioned above. Most MPI implementations such as MPICH-

G2, MPICH, MPICH-V2, and GridMPI[37] directly manage the connections of a static network.

An overlay network is a virtual network constructed on top of the static network. The overlay network can resolve communication issues such as NAT and firewall problems at a lower layer, and provides a virtual address system to a top layer to be used by the nodes. JXTA™ [38] provides an address system to the nodes using its protocols. Another advantage of overlay networks is that a computation node can be separated from the physical computational resource. By moving the data that represents the computation node with its virtual addresses, the computational resource can migrate to another processor. The uses of network overlay for the Grid include getting Grid nodes to communicate about Grid management services, and uses in parallel computing as is done with P2PMPI [39]and P3 [40]. Disadvantages of overlay networks is that the messages may take longer to propagate depending on the physical topology, and they can add packet overhead.

Load Balancing: The use of load balancing is important on distributed memory algorithms. The concept consist of balancing the work by sending more jobs from processors that are overloaded with work to processors that are idle, or are processing the work-load faster. Skeleton-based frameworks do not consider synchronized programs, so the information is always running in a pipeline or work pool manner. This gives them the choice to treat all the data as packets and the load is balanced by sending more packets to the processors that are idle. Pattern-based frameworks rely on the pattern designer to provide load balancing features to the user programmer. MPI, and OpenMP do not provide load-

-balancing features; for those frameworks, is up to the programmer to localize potential load imbalances and account for then in their code.

Latency: Latency is the time it takes a data packet to get from one node to the other. There are two principal techniques to reduce latency. One is to hide it, which consist of finding something to compute to keep the processor busy while the transaction is being done. The other technique is to send data in big chunks so that the number of times the latency penalty is incurred is reduced. There are other ways to combine latency hiding and sending data in big chunks into a hybrid technique [41]. The projects reviewed for the literature search did not have latency coping mechanisms, but as mentioned with load-balancing, it is possible to include the features on skeletons, and patterns.

Bandwidth: Bandwidth is the number of bits per second that can be transmitted through a link. There are techniques to reduce bandwidth's influence. Two of them are: compressing the data before is sent over the link, and reducing the total data sent by redundantly processing the same values at the computers that need them. This works by having redundant copies of the input data necessary to compute the output data at the nodes. The projects reviewed from literature do not include techniques to cope with the bandwidth, although, like latency, the techniques could be included with skeletons and patterns.

Fault Tolerance: Fault tolerance consist of preventing the parallel application from failing when one or more of the computing nodes fail. This is more likely in Grid computing because there are more computing nodes involved, and there are differences among institutions on how reliably the resources are maintained. The techniques used to

provide fault tolerance are: check-pointing, message logging [42], and redundant use of nodes [39].

1.2.4 Conclusion

This literature review presented a high-level view of programming for the Grid environment from many different perspectives. The programmability focused was on programming language features, extensions, and automatic code generation focusing on the role the programming language plays in development. Multiple frameworks from industry and academia where described for their most relevant features in relation to a heterogeneous environment. Section 1.2.2 described the programmability and performance issues in using shared and distributed memory and one approach that has been used to resolve it on the HPC field. The last section focused on some of the non-functional concerns such as load-balancing, fault-tolerance and connectivity, and the pertinent framework that implemented solution was discussed. The review should have provided a perspective on the multitude of technical and academic problems that phase parallel applications in the Grid. The review of the skeleton/pattern approach and the frameworks that implement it will be used as parting ground for the goals proposed in the next section.

1.3 Research Contribution

The research involved one main goal, and the subsequent contributions were derived around the main goal. The main goal was to extend the skeletons/patterns parallel programming approach to increase its feasibility for a wider population of application and research programmers. The specific contributions are:

1. The pattern adder operator is proposed: Even though there are many skeletons/patterns, some literature suggests there is a need for more patterns to ad-

dress less typical algorithms. Instead of adding new skeletons/patterns, work was done to develop the a pattern adder operator and measure its benefits. The work reduces the population of patterns while at the same time giving the programmer better tools to create complicated patterns out of the basic ones. The concept was implemented and tested. More on this contribution is found in Chapter 3.

2. The hierarchical dependency concept is introduced: The hierarchal dependency concept is used to allow for automatic scalability. Automatic scalability means that a pattern can expand the number of processors used during run time to achieve optimum performance. First a streamed data flow programming technique was implemented into the Seeds framework. The hierarchical dependency was added to the stream data flow technique. The hierarchal dependency concept was implemented and tested. Additionally, a synchronized process hibernation protocol was designed as part of the automatic scalability goal. More on this contribution is found in Chapter 4.
3. Support for asynchronous scheduling on the Seeds framework: Because of its use of peer-to-peer connectivity, the framework can start a parallel program with a partial number of the nodes on-line. As more nodes come on-line, the framework can allocate work to the new nodes. The synergy between the hierarchical dependencies and asynchronous scheduling can be used to start jobs when only a small set of processors are available.

1.4 Scope

The research was focused on scalability, adaptability, and programmability. There are other neighboring topics in this field that were not researched:

Scheduler: The framework will assume that a scheduler or meta-scheduler is provided. Scheduling jobs on the Grid is a topic that is still being researched because of the NP complexity of mapping parallel programs to the Grid. The resources given to the framework are assumed to be available.

Security: The services related to security are assumed to be resolved by a lower layer. For the framework, this is accomplished by the Globus Toolkit [36] or Secured SHell (SSH)[43], which provides job submission, file transfer and security features. The security features include the verification of the resources as well as the verification of the user, and the optional use of encryption.

Fault tolerance: was no explored in this research and can be part of future work.

1.5 Impact of Work

This research creates procedures, algorithms and software tools that help facilitate software development for all platforms, with an emphasis on the Grid environment. At the high end of expectations, the ideas studied by this work can reduce development time at the cost of a small performance lag. The tools provide services that are needed to make the Grid resources competitive with cluster computers and pools of workstations with a small learning curve on the side of the programmer. The tools can easily play a roll on cluster computing and cloud computing as well. The tools can increase the use of resources, and improve productivity.

1.6 Conclusions

The Grid remains an environment where the development is complicated. Multiple lines of code need to be written to account for the behavior of the medium. The extra conditions that require more complex code usually deter programmers from running an appli-

cation on the Grid. Institutions could integrate multiple clusters to confront bigger problems, but the current software infrastructure makes the prospect labor intensive. The presented research extends the skeleton/pattern parallel programming approach in order to increase its used by researchers and industry.

Chapter 2 describes the Seeds framework created for this research. The Seeds framework can be considered the laboratory where the research extension were implemented, tested, and measured. Chapter 3 proposes the pattern adder operator. The operator can be used to combined patterns to reduce the creation of new patterns, and to reduce development time. Chapter 4 introduces the hierarchical dependency concept. The concept uses data flow programming and a technique explained in Chapter 4 where the framework gains back control from the pattern after every iteration on the computation. The combination of these concpets plus other algorithms can enable a pattern to automatically scale on a heterogeneous environment. Chapter 5 presents results on modifications to the framework to improve performance. Chapter 6 describes the future work, and Chapter 7 summarizes the main ideas and contributions from the dissertation.

CHAPTER 2: THE SEEDS FRAMEWORK

Starting with the concept of a computational Grid, we have set out to envision a future parallel computing environment, its potential, and its shortcomings. The main advantage of Grid and heterogeneous environments such as the Cloud and multi-core environments is their increasing computational resources. However, the heterogeneous environments have the same challenges that previous generations of parallel programmers encounter when developing solutions for cluster and super computers. In addition to that, the newer environments have to cope with more complex conditions, such as heterogeneous networks with dynamic changes in bandwidth and latency, and changes in the number of processors available scheduled by different agents. The Seeds framework can be deployed into such an environment, and coping with those non-functional concerns are research topics that can be tackled using the framework. Seeds has been used to implement prototypes on programmability topics, and on performance topics:

- Pattern extension[44]: We have implemented forms to extend parallel pattern programming to make them a more viable choice for parallel programmers. One of the challenges for patterns is when a programmer does not find the pattern needed, and a new pattern may have to be created from scratch as a result. One possible solution implemented on the framework is to get patterns to interact in order to create more complex patterns using simple patterns. The concept reduces

the need to code parallel applications at a lower level where communication lines have to be managed. Refer to Chapter 3 for more on pattern operators.

- Automatic Scalability: Scaling an application to use more resources could be done dynamically on a heterogeneous environment. Given the dynamic nature of the Grid, the number of computation units can increase during run time because of how resource discovery works, or because job schedulers may not synchronized with each other when a grid application is launched. A dynamic auto-scalability feature was extended on top of pattern programming. Refer to Chapter 4 for more on auto-scalability.

The rest of the chapter is organized as follows: Section 2.1 presents the general list of features implemented into Seeds. Section 2.3 explains what the JXTA platform is, its main features, and how the platform was used in the implementation of Seeds. Section 2.5 shows important technical details that concern with the abstraction of communications between processes in a Grid environment. Section 2.6 explains how the processes are managed in a multi-core computer, and how the parallel techniques are implemented into the framework. Finally, Section 2.10 presents results that compare the Seeds' performance against other production level parallel frameworks.

2.1 Framework Overview

There are many parallel frameworks both in production environments and in the research field. Most of these frameworks are mention in Chapter 1. There are some features that were not present in any other frameworks at the time we started development. The features are needed in order to research the aspects into Grid parallel computing. One feature is a dynamic number of processes coming in and off line during runtime.

MPI, and similar frameworks where static, and therefore were not good starting points for the research. Also, from the programmability perspective, adding new processors during runtime can be unnecessary complicated on MPI due its use of numbers to map a processes to processors.

The need to adapt to heterogeneous networks such as those with NAT and firewalls was also seen as necessary for a Grid-enabled parallel computing platform. This communication adaptation was done on other frameworks, namely Ibis [35] However, the pattern based features were not present in the Ibis project. Other frameworks use the Remote Method Invocation (RMI) protocol to enable communication for their high performance parallel environment[9]. However, the RMI protocol relies purely on a data pull method, this means the client process requests for the data. The method can incur the round trip time delay (RTT) twice in comparison to a the synchronized communication used by MPI and basic socket connections.

Initially, the need for a language that would resolve the problem of heterogeneous hardware was considered high in the priority list, but the importance of a JIT language is not seen as important at this point as the Cloud and multi-core clusters provide homogeneous architectures that can scale up to thousands of cores. This means that an architecture independent language is a requirement only for the Grid environment. Cloud computing and cluster computing do not have the problem of dealing with a highly heterogeneous computing environment.

The Seeds framework has been created to serve as a platform for the ideas and techniques and implements multiple necessary features for a parallel program that can run on

top of Grid middle-ware. A home page is maintained which provides software developed, complete documentation, and tutorials[45]. The framework is called Seeds because it deploys a seed folder that "grows" into a network capable of running a parallel program. The main characteristics of the framework are:

- Language: The language chosen for the framework was Java. This JIT Language can be run on multiple types of hardware. This eliminates the problem of porting the framework to different architectures and having to manage the libraries used by programmer. Java also enjoys ample support from the Globus Tookit and multiple distributed computing platforms.
- Topology: The framework has been implemented using moderate use of peer-to-peer (JXSE) networking. Peer-to-peer provides the overlay network that introduces a layer on top of the hardware network that makes it possible to refer to a process independent of the hardware host.
- Programming style: The framework uses the pattern programming style. The top layer leaves an environment for parallel programs that is simplified. On the bottom layer, the framework provides multiple services related to scalability.
- Memory management: The framework uses message passing and takes advantage of shared memory. If the processes are on a multi-core processor, they share a queue on shared memory.
- Network connectivity: The framework provides gateways for the hosts behind a NAT.
- Self-deployment: The framework can be self-deployed. This is provided because

the Grid nodes collaborating in the computation may not have the framework available. The framework deploys using the Globus Toolkit or SSH. The framework uses the Java Cog Kit to have access to the Globus tools. The user needs to provide the local path to the shuttle folder, which is the folder that contains the third party Java libraries: JXSE [38], UPNPLib [35], and Seeds libraries.

- Interfaces: The interfaces and abstract classes are used mainly to impose framework rules on the advanced and basic users (see Section 2.8). Because the modules are inserted in the middle of the execution code, there needs to be some guidelines on what the user's application should handle. To do this automatically, all the tasks necessary to keep up with framework requirements are handled by abstract classes. Abstract classes allow us to have variables and implement functions while at the same time leaving some functions to be implemented by the user. Interfaces, on the other hand, can only specify signature methods.
- Adapts to dynamically changing processes: the framework adapts to a changing Grid environment where the number of processes is not constant. Not all the processes are available at startup, and some processes will drop during runtime but not due to failure. The cases where this happens is in Grid nodes where the scheduler are not coordinated to start the parallel application at the same time. Other situations that add dynamically changing processes are the administration of processes based on priority on a busy cluster or for energy saving on a multi-core computer. In a multi-core computer, a load balancer can shrink a parallel application to use less resources more optimally shutting down the idle remaining cores.

2.2 Dropped Requirements

Some feature were explored but dropped because of poor performance:

- Automatic topology configuration: Automatic topology construction was eliminated from the list of requirements. The topology construction requirement adds overhead in terms of starting up the framework, and the algorithms devised have proven unreliable. The nodes are divided into leafs and directors. The leaf are computing processors. The directors compute like the leafs, reroute message to organize the network, and can provide a gateway connection to the leafs. Because JXTA's has limit on the efficiency for the number of rendezvous nodes [46], the number of rendezvous is kept low to just one per Grid node. The algorithms have consisted of organizing the nodes by using multicasting packets, and a shared file partition. Each node writes a hash number derived from the host's name, and the host with the lowest number is selected to be the leader. The algorithm with multi-casting requires a few seconds, from one to three seconds on a cluster environment to be effective. Unfortunately, some clusters have multi-casting disabled, so the algorithm has to be repeated using other communication approaches. The other communication approach attempted was using a shared file system. This is more effective since all hosts within a Seeds Grid node have to share the file system. But this approach also has some shortcomings. Because the processes come on-line at different times, the algorithm also needs one to three seconds to increase the probability that it will be effective at selecting a leader, and the by-product files that are left from a previous run can hamper future runs. For these reasons, the framework was modified to have a statically set leader selection, and

the automatic selection of a node leader is left as future work. In this case, the user would be responsible to determine a node that has the most unrestricted access to the highest level network (usually the Internet).

- Firewall circumvention: Another requirement that is now viewed as impractical is circumventing around firewalls. This is mainly because the algorithms to find open ports on the Grid nodes are too slow compared with the high performance expectations for parallel programs. As a result, a virtual organization must at least open a few ports in the 50,000 to 51,000 range, in order to be able to run Seeds.

2.3 JXTA Platform

JXTA is a peer-to-peer (P2P) platform originally create by Sun Microsystems. Its messages are propagated through the network using a Loosely Consistent Distributed Hash Table (LC-DHT) algorithm [46]. Antoniu et al. give performance results that show JXTA can be used for high performance computing in Java [47]. Initially, the choice for JXTA was to take advantage of many network abstraction protocols. However, as development continued, some of the features were too slow, or not ideal for high performance. Antonium et al. points out the JXTA Socket have higher delays than Java socket. Given that Java sockets already add delay over native sockets, we decided to implement the high performance pipes using Java sockets. JXTA also provides NAT circumvention tools, but they rely on the HTML protocol, which add overhead to the connection. We instead implemented a TCP re-routing alternative. As a result, the JXTA platform is used mostly to create and distribute advertisements, to create uniquely identifiable ID's using a data structure, and to organize the network of nodes around the user's application. The

rest of this section explains important details about JXTA, its primary use in the JXTA community, and how the concepts are used in relation to Seeds.

2.3.1 Node Type

The framework uses the JXTA platform to manage communications. In JXTA, the peers can have different categorizatios that indicate the type of network that is being constructed with them [48]. The categories are:

- Ad-hoc: Used to create a decentralized peer-to-peer network.
- Edge: A peer that is on the edge of the network, and is not required to route messages to other nodes.
- Rendezvous: A peer intended to route messages to other peers as well as do all the tasks an edge node would do.
- Relay: This node will provide an HTTP communication alternative for nodes that are behind NAT routers. The protocol is provided as HTTP to also circumvent firewall's filtering.
- Proxy: Provides connection to JXME node. JXME nodes are peers running on small mobile devices such as cell phones.

Out of this list of categories, the Seeds framework only uses rendezvous, and edge. Seeds divides the Grid into Grid nodes. Each of the Grid nodes is made up of a computer cluster that shares a file system.

- DirectoryRDV: In a Grid Node, there is only one rendezvous node that is called a DirectorRDV, and it is managed by a Java class with the same name. This node is selected to provide a gateway for LeafNodes in a cluster without access to the

wider network.

- LeafWorker: The other nodes in the cluster are called LeafWorkers and are assigned the JXTA category Edge. This nodes are mainly workers, they only wait for new jobs, execute them, and idle for the next job.

2.3.2 Advertisements

Advertisements in JXTA are XML documents that are exchanged with nodes on the network. The advertisements are used by JXTA protocols to announce new services, and communication pipes at different levels of the platforms communication stack. Advertisements are also important for Seeds because they are used to distribute information about a pattern as well as information about communication lines. The main advertisements used by Seeds are:

- DataLinkAdvertisement: This advertisement is used as part of the multi mode pipe communication layer. It provides information necessary to create the least overhead connection between two nodes. The three alternatives are shared memory, socket, or through a gateway. Description for the attributes in this advertisement are found in Section 2.5.
- SpawnPatternAdvertisement: This advertisement is used to start a new pattern on the peer-to-peer network. The advertisements' main attribute is the name of the parallel pattern to be executed and an optional list of arguments. More on this advertisement's attributes is in Section 2.7.
- DependencyAdvertisement: This advertisement provides information to create a connection using the data flow model of parallel programming. The model allows

the framework to provide automatic scalability to the parallel pattern using hierarchical dependencies (Chapter 4).

- NetworkInstructionAdvertisement: This advertisement allows the network to have some of the services needed from a distributed operating system. The advertisement can be used to send specific instructions to the nodes while they are executing the parallel application. The main purpose for the instructions is to preempt the parallel application and shutdown the network.

2.3.3 The Protocol Layers

JXTA has six protocols [48] They are the Peer Resolver Protocol (PRP), the Endpoint Routing Protocol (ERP), the Peer Discovery Protocol(PDP), the Peer Information Protocol(PIP), the Pipe Binding Protocol(PBP) and the Rendezvous Protocol (RVP). Most of the protocols are self explanatory in their function, and more information about them can be found at [48]. The Seeds framework uses only the Message Propagation Protocol, which is part of RVP, which relies on ERP. This protocol is used by means of publishing and queering for advertisements. All other protocols were avoided in favor of direct socket connection. JXTA's layers perform better for high throughput tasks, while having large latencies[47].

2.3.4 JXTA ID

JXTA has a peer-to-peer identification specification that is used to identify peers, peer communications, and advertisements. Figure 2.1 shows a JXTA ID Universal Resource Identifier (URI).

```
urn:jxta:uuid-
59616261646162614E50472050325033189386 42E4714B929774F6E709C8EADF04
```

FIGURE 2.1: JXTA ID example.

Because the number of possible JXTA ID's is very large ($4.7x10^{80}$), creating the identification number randomly has a low probability of conflicting with an ID created by another node somewhere in the network. Because of this, the Seeds framework also uses the JXTA ID to define the pattern ID's that identify the parallel application and prevent the framework from mixing communication lines of two or more patterns running at the same time. The JXTA ID is also used to identify the multi-mode-pipe communication lines implemented for Seeds.

2.4 Starting up the Framework

The process starts at the user's workstation. The framework loads and starts by sending the shuttle folder to the Grid nodes present in the Grid node file. Once the files are transferred, the framework submits job tasks to the grid nodes to boot up the network. Once the network is up, the nodes get organized and start running the user's module. The process continues until the task is done, an unrecoverable exception happens, or the user stops the network by issuing a shut down advertisement.

Once the processes start running, each node starts with a class called the Node class. This class is responsible for surveying its network environment. Based on the survey results, the Node class determines the network's role for the node. The Node class determines the node's role based on the network type detected. Once the Node has determined the network type and role for the node process, it cedes execution to either class LeafWorker or class DirectorRDV.

2.5 Communication Management

Seeds provides an abstracted communication layer to its advanced layer. The objective is to transparently provide a connection between two processes in the Seeds network without transferring the non-functional concerns of where the remote process is located. A similar abstraction has been done by Maaseen et al. [34]. The communication mainly deals with three types of communications: Shared, Distributed, and NAT. Shared memory will share a queue between the two processes that need to have a connection if both of the processes are located on the same multi-core processor. As studied by Henty, some algorithms do not gain any advantage by programming directly to share memory compared to a share memory implementation of message passing [49]. The distributed version of the communication manages a socket that is used to communicate two remote processes. The NAT communication uses a gateway to allow a process that is inside a NAT to communicate with the rest of the network as if it was also on the more open network. This feature is implemented with high overhead though. The feature is not used on any research experiment relating to programmability improvements. Figure 2.2 shows performance results of the three communication options. Shared memory performs best because of the lack of overhead incurred by the layers of networking. The WAN connection, which is a socket connection, shows acceptable performance as well. Showing lower performance are NAT-WAN, WAN-NAT, and NAT-NAT connections. All the NAT circumventing tests had an extra process posing as a gateway node, and all the tests were done in a multi-core computer to avoid introducing overhead due to latency and bandwidth. The experiment involved transferring one thousand objects over the connection with different amounts of double data types. In the figure, the *x* axis shows the number

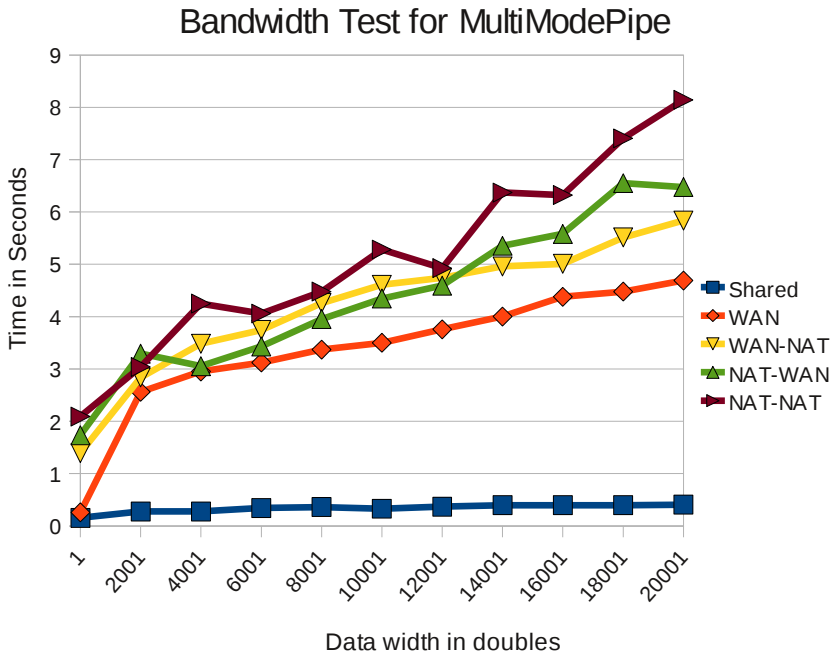


FIGURE 2.2: A preliminary bandwidth test on the MultiModePipe communication line.

of doubles sent. The *y* axis shows the time taken to transfer all the objects. Each test was done five times to reduce noise. The test was performed on Coit-grid Shared Memory for which more configuration details are provided in Section 2.9.

2.5.1 DataLinkAdvertisement

The communication abstraction is called a MultiModePipe. The MultiModePipe was implemented following Java's implementation of a Socket. It has a MultiModePipeDispatcher, which is similar to a SockerServer. It has a MultiModeClient, which is similar to a Socket class. And it has a Connection Manager class which is similar to a Stream class. The node that starts the communication uses the MultiModePipeDispatcher. The dispatcher starts the Socket and shared memory dispatchers. The client then uses the class MultiModePipeClient to get a connection to the dispatcher. The connection is established using the information shared through a DataLinkAdvertisement. Figure 2.3 shows

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:DataLinkAdvertisement>
<jxta:DataLinkAdvertisement xml:space="default">
        <Name>
                Data Link Advertisement
        </Name>
        <DataIDTag>
                4
        </DataIDTag>
        <GridNameTag>
                GridTwo
        </GridNameTag>
        <WanOrNatTag>
                NAT_UPNP
        </WanOrNatTag>
        <DataLinkPipeIDTag>
       urn:jxta:uuid-59616261646162614E50472050325033345 11DA301284F0695FD8D93FF0D066C04
        </DataLinkPipeIDTag>
        <RoutedDataLinkPipeIDTag>
       urn:jxta:uuid-59616261646162614E50472050325033C2846BEAC999466994B6EA8137DEA60404
        </RoutedDataLinkPipeIDTag>
        <LanAddressTag>
                192.168.0.100
        </LanAddressTag>
        <WanAddressTag>
                97.89.111.142
        </WanAddressTag>
        <PortTag>
                50091
        </PortTag>
        <PatternIDTag>
       urn:jxta:uuid-59616261646162614E50472050325033808B2CE699114AEF895EB2B839DAC78304
        </PatternIDTag>
</jxta:DataLinkAdvertisement>
```

FIGURE 2.3: DataLinkAdvertisement example.

an example of a DataLinkAdvertisement.

The attributes created for the MultiMode connection are:

- Name: A unique name to distinguish the advertisement from other advertisements.
- DataIDTag: This integer is used to identify the processes announcing the connection.
- GridNameTag: The name of the Grid Node. This is needed to determine the Network Interface Card (NIC) to be used to communicate with a remote process.
- WanOrNatTag: This announces the type of network where the emitting process is located. This is also necessary for remote processes to determine the type of connection that can be established with the emitting process.
- DataLinkPipeIDTag: A JXTA ID assigned to the communication. In contrast to the DataIDTag, this ID will be different for every connection.
- RoutedDataLinkPipeIDTag: This JXTA ID tag identifies the unique ID of a gateway node that can be used to route messages in case the emitting process is behind a NAT router.
- LanAddressTag: This tag has the IP address for the emitting process. The address is to be used by nodes that share the same Grid Name, and therefore are in the same network.
- WanAddressTag: this tag is used by the remote processes that connect to the emitting process. The address can be the Internet address of the emitting node, or the Internet address of the gateway.

- PortTag: Port used by the node.
- PatternIDTag: a unique JXTA ID identifying the pattern. This is used to prevent the patterns from jamming each other's connections when more than one pattern is running. This can happen because the DataID is used to map a connection advertisement to the node that provides that connection.

Upon receiving the advertisement, a client must decide which type of connection is optimal. Figure 2.4 shows the decision tree for the MultiModePipe connection algorithm. The DataLinkAdvertisement has important information about the emitter that will be used in the decision process. Most of this information is gather during the network's start up process. Seeds has three categories:

- WAN: Wide area network. The node will contact a server on the Internet. If the IP address reported by the server is the same as the IP address returned by one of the Network Interface Cards (NIC), the node is in a WAN.
- NAT_UPNP: Network Address Translation with Universal Plug N' Play is as accessible as WAN. If the IP address from the Internet server does not match any of the NIC's, an algorithm checks for UPNP NAT routers. If one is found, this category is assigned.
- NAT: The node is behind a Network Address Translator router. The node will only be able to communicate to other nodes within the network, or through a gateway. This status is found if the IP conforms with standard RFC1918 [50], and queering for a UPNP NAT returns null.

To establish a connection, the client first checks a static map. The map logs informa-

tion for shared memory connections created by the processes residing in the same multi-core processor. Traversing the decision tree in Figure 2.4, we have the following connection routes:

- If the connection ID returns not null, the server process is on the same processor and a shared memory connection can be established.
- If the server is WAN and the client is not NAT, the connection is done with a plain socket.
- If the client is NAT, it connects to the server using its Grid node's gateway.
- If the server is NAT, the client checks if the Grid node name is the same. This would mean that both hosts are on the same cluster, and therefore can connect using a plain socket through the LAN address.
- If the Grid node names are different, the client checks if it is inside a NAT as

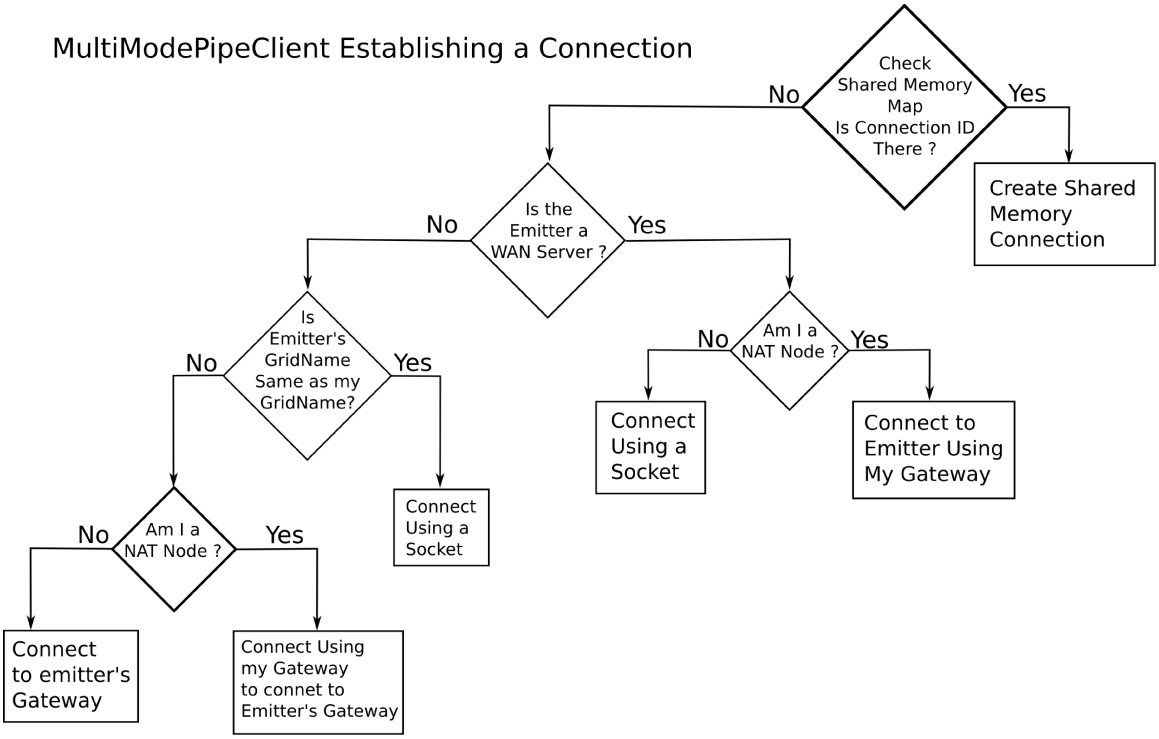


FIGURE 2.4: Decision tree from the perspective of a client process establishing a connection to the connection server.

well. If it is, the client connects to the emitter's gateway using its gateway.

- If the client is not a NAT node, it connects using a socket to the server using its gateway. It should be noted that the Grid node name is configured by the user at the AvailableServer.txt file before deploying the framework.

Seeds packet communication: The framework uses Serializable interface objects as the primary way to send and receive packets. This adds a layer of overhead to the framework. Measurements done show a least 22 bytes of overhead for generic Java objects. A String object serializes with less overhead presumably because of optimizations specific to the String object. Ideas on reducing this overhead a presented in Chapter 5. Other parallel frameworks such as Mapreduce[7] use objects to package data. It is unknown at the time of writing whether MapReduce optimizes serialization.

2.6 Process Management

Seeds manages the processes around the user's application, The framework organizes the nodes so that one node can spawn a new pattern, and the other nodes can react to the pattern instruction and construct the communication pattern needed for the application. This section explains how the framework handles multiple processes on a multi-core computer, how a pattern advertisement is turned into a series of classes that will load initial data and start running a parallel application, and finally, the use of a three layer development technique is explained.

2.6.1 Handling Multi-core Hosts

The process, whether it is a single-core remote host or in the local machine through multi-core, are managed using Java threads. One thread is created for every core in each of the machines. The class Worker is responsible to manage those threads. The frame-

work can run multiple patterns at the same time. The Worker class will receive the pattern advertisement as they are caught by JXTA. If a pattern that has not been executed is received, the Worker will proceed to check if there are idle threads available for the new job. An exception is made if a sink/source node is requested. In that case, a thread is allocated independent of how many processes are already running on the node. The sink/source node is given preference because the parallel pattern cannot run without a sink/source node. Once the pattern finds a thread, the Worker will deploy the thread, and execution is relinquished to that thread. The thread will run one of the user's remote processes. Once the process is done, the thread dies, and the Worker class counts that thread as idle again. A low priority thread is run on the background to continuously scan the JXTA network for new advertisements for new communication lines, new patterns, or a network instruction that can take precedence over the user's module.

2.7 Parallel Techniques Implemented in Seeds

There are multiple documents and tutorials at Seed's website that explain how to use the framework to implement a parallel pattern [45]. This section discusses how the framework converts the user's module, and the advanced user's pattern into code that is run on the remote hosts. The user deploys the pattern using Seeds.startPattern() method. Seeds creates a pattern advertisement from the user's module. Figure 2.5 shows a sample of the SpawnPatternAdvertisement. The attributes for the SpawnPatternAdvertisement are:

- PatternClassNameTag: This has the user's full Java class name. This attribute is used, together with Java abstractions API, to instantiate the user's module at the remote processes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:SpawnPatternAdvertisement>
<jxta:SpawnPatternAdvertisement xml:space="default">
        <PatternClassNameTag>
                edu.uncc.grid.seeds.example.stencil.HeatDistribution
        </PatternClassNameTag>
        <GridNameTag>
                GridTwo
        </GridNameTag>
        <PatternIDTag>
         urn:jxta:uuid-59616261646162614E5047205032503380 8B2CE699114AEF895EB2B839DAC78304
        </PatternIDTag>
        <SourceAnchorTag>
                Kronos
        </SourceAnchorTag>
        <SinkAnchorTag/>
        <RemoteArgumentsTag/>
</jxta:SpawnPatternAdvertisement>
```

FIGURE 2.5: A SpawnPatternAdvertisement XML document sample.

- GridNameTag: The Grid name for the emitting node.
- PatternIDTag: The unique JXTA ID used to identify the pattern. The pattern ID helps the framework differentiate communication lines, and this in turn allows for features such as nesting patterns and skeletons, as well as the implementation of pattern adder operators covered in [44].
- SourceAnchorTag: The tag provides the host name for the server that should have the source and sink nodes for the pattern. Since all patterns need to have a starting point for the initial data, and must converge at some point for the final result. The source and sink have to be at the same host for the current implementation. Not providing an Anchor defaults the Anchor to the host where the pattern is being spawned.
- RemoteArgumentsTag: This tag is used to pass string arguments to initialize the remote instantiations of the user's modules. This feature is similar to the arguments list provided by the main method for an executable Java class.

The advertisement is distributed on the peer-to-peer network. Once the remote nodes catch the advertisement, the framework gets the module from it, which is used to get the Template class. The module class is part of the basic layer, and at this point, the framework loads the classes that will manage the advanced layer. The layer distinction allows the framework to provide more complex tools of development to the advanced user, but it also allows the expert layer to provide the parallel tools using different parallelization techniques to the advanced user.

There are three implementations for Seeds parallelization techniques: message passing (MPI-like), socket-based and the data flow. Note that only pattern/skeleton programming and the data flow technique are of interest to our research. Message passing is implemented to use as control for programmability experiments, and socket-based are implemented because it is the lowest level abstraction for a two point network connection.

Figure 2.6 shows the relationship between the different parallel communication techniques implemented within Seeds. The yellow label shows the skeleton and pattern implementations. The light blue labels show the parallel programming techniques. As a rule of thumb, it takes more lines of non-functional code to create the same program done with skeleton/patterns if done with sockets, message passing (MPI), or data flow techniques [19].

- Unordered Template: In an unordered template, the number of processes can vary during the computation. These new processes can be added or subtracted without much change to the original parallel implementation. The main example for this is the workpool skeleton. The template is implemented using the UnorderedTem-

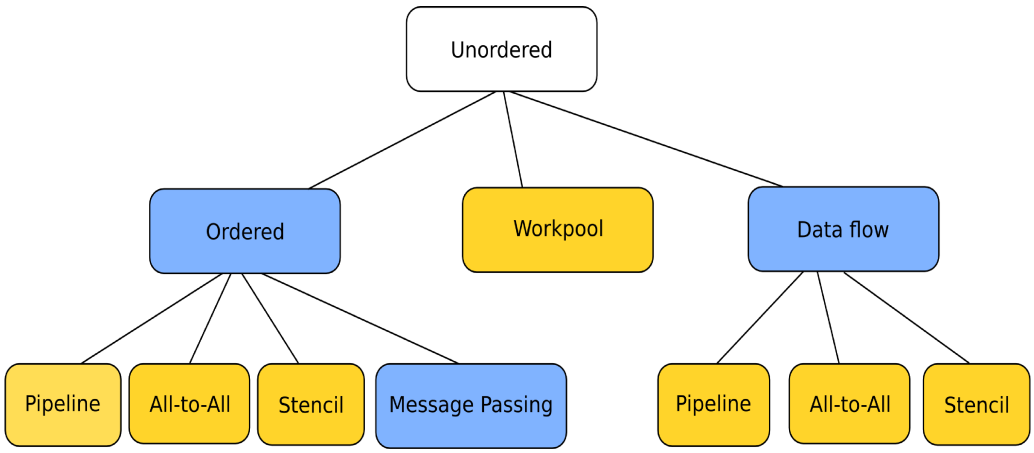


FIGURE 2.6: Unordered, Ordered, and Data flow parallel programming techniques implemented into Seeds.

plate class. The framework proceeds to execute the code on that template, which is developed using the socket-based technique. When the template is done executing, the control flow returns to the framework, which idles waiting for a new pattern.

- Ordered Template: The ordered template has a fixed number of processes from beginning of computation time to the end. If an ordered template needs to run, the template first needs to request a Loader object. The loader will ensure all the remote processes have the initial data before the parallel application starts. Examples of Ordered templates are the pipeline skeleton, the stencil, and the all-to-all patterns. The loader is an unordered template that will restrict the environment to run the order template.
- Message Passing: A message passing template is also available. This pattern inherits from the ordered pattern because, like MPI, it needs all the processes to be on-line before the parallel application is started. The pattern is MPI-like and not an MPI implementation because the actual MPI or MPJ standard was not fol-

lowed. A loader is called to constrict the environment similar to how it is done for the ordered template.

- Data flow: The data flow consists of data flow processes that we call perceptrons. Each perceptron has a list of inputs and outputs, and a computation method. The advanced user creates patterns using data flows by designing the number of perceptrons used. The communication pattern is created by assigning inputs and outputs to the perceptrons. The data flow is deployed by a loader. However, the Loader can add or subtract more data flow units during run time to expand or contract the computational resources as the parallel application computes. More on how the data flow works can be found in Chapter 4.

2.8 Three Layers of Development in Seeds

The framework divides its programmers into three groups: expert, advanced, and basic. This is a feature borrowed from CO_2P_3S[14]. Figure 2.7 shows the order and name of the framework layers. At the bottom, there is the JXSE and Globus/CoG layers, which are the libraries used to boot-strap the network. Then, the expert layer creates a simpler environment for the advanced layer, and the advanced layer provides skeletons and patterns to the basic layer.

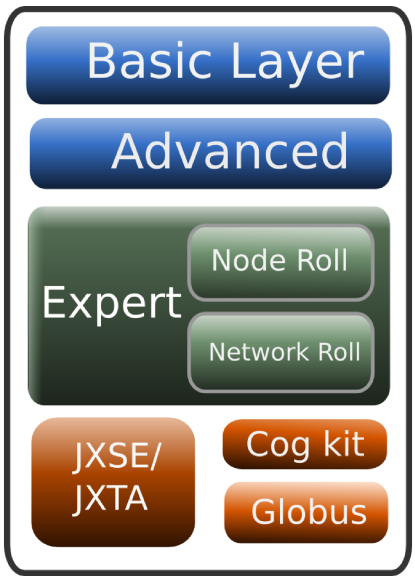


FIGURE 2.7: The three layers of development implemented into the framework.

2.8.1 Expert Layer

The expert layer is made in modules that make it easier to re-code or improve pieces of the framework. It is divided into the deployment phase, P2P and network phase, the communication phase, and the parallel programming technique to be used by the advanced layer. The main task to be accomplished by the expert layer is to turn a pool of heterogeneous resources, heterogeneous networks, and other different concerns into a pool of somewhat organize resources that behave in a manner that can be compared to an MPI environment or a data flow environment. The advanced layer picks up the execution on top of the abstracted environment created by the expert layer.

2.8.2 The Advanced Layer

The advanced layer is considered a layer where communication lines have to be managed. In the basic layer, the pattern already provides a template on how the communication and computation tasks are managed, but in the advanced layer, the programmer must

resolve those tasks. Therefore, this layer is exposed to deadlocks and raise conditions that are common in message passing and thread-based parallelization techniques. As explained above, the advanced layer can be programmed with one of four parallel programming techniques: UnorderedTemplate, OrderedTemplate, Message Passing, and Data flow:

- UnorderedTemplate: The advanced user is given two signature methods. The ServerSide method is used by the source/sink node, and ClientSide method is used by the worker nodes. Figure 2.8 shows the interface. The advanced user is to create socket-based connections, preferably using the MultiModePipe, and create the communication and computation tasks.

```
public abstract class UnorderedTemplate extends Template{
      public UnorderedTemplate(Node n) {
            super(n);
      }
      public abstract void ServerSide(PipeID pattern_id);
      public abstract boolean ClientSide(PipeID pattern_id);
}
```

FIGURE 2.8: The UnorderedTemplate interface.

- OrderedTemplate: This method has a loader interface and a computation interface. Figure 2.9 shows both of the interfaces. The computation only manages the computation nodes, the loader interface (bottom) is used by the framework to request initial data to the advanced user. The initialization packet is sent to the remote nodes, and the computation is started using the computation interface. The main signature methods are the DiffuseDataUnit to get the initialization data and the GatherDataUnit used to return the processed data. GetDataUnitCount is used to get the number of processes needed for the program, and instantiateSourceSink

```
public abstract class OrderedTemplate extends Template{
      protected long CommunicationID;
      public OrderedTemplate(Node n) {
            super(n);
      }
      public abstract boolean ComputeSide( Communicator comm);
      public abstract void Configure( DataMap<String
                              , Serializable> configuration);
      public abstract Class getLoaderModule();
}
public abstract class PatternLoader
                              extends BasicLayerInterface{
      public static final String  INIT_DATA = "init_data";
      protected OrderedTemplate OTemplate;
      protected PipeID PatternID;

      public Data Deploy(DataMap input){
            ...
      }
      public abstract DataMap DiffuseDataUnit(int segment);
      public abstract void GatherDataUnit(int segment
                                          , Data dat);
      public abstract int getDataUnitCount();
      public abstract boolean instantiateSourceSink();
}
```

FIGURE 2.9: The OrderedTemplate interface.

is used to tell the framework if there needs to be an additional process feeding data after the parallel program has started. This is particularly needed in streaming implementations such as the pipeline skeleton. The Stencil, and all-to-all patterns do not make use of this feature, so the user returns false on this method. The OrderedTemplate (top) interface's main method is ComputeSide. The computation function is only partially controlled by the advanced user, the framework regains control after each iteration around the main loop. We assumed all algorithms implementing this interface will have a main loop. The requirement is added in order to enable the pattern adder operator feature that is discussed in [44]. The advanced user uses the Communicator class *comm* to get connection lines with other remote processes including processes in the same processor. The framework will create the connection between two processes utilizing the data ID

that is distributed on the network using the MultiModePipe advertisement. The connection is established when the first packet of information is sent by a process. That is, the communication line is not established until the a packet is sent from one process to the other. The communication line stays on throughout the life of the parallel application once connected.

- The Message Passing Template: The message passing template inherits the qualities from the Ordered template. The interface used is the same as the one shown on Figure 2.9. The main difference is that the message passing implementation does not have to return the control to the framework after every iteration.

2.8.3 The Basic Layer

The last programming layer is the basic layer. The basic interfaces are programmed following pattern parallel programming techniques. The basic user finds a parallel programming pattern that best describes the problem's computation and communication characteristics, and based on that, chooses the appropriate basic interface.

There are many interfaces implemented for the Seeds framework. Figure 2.10 shows an example of for a pipeline. In it, the basic user is to fill in the computation, diffuse, and gather methods to implement an application that will take advantage of the communication pattern provided by the pipeline. The user, in the computation method, would then compute the appropriate stage based on the integer number *stage*. The method getDataCount should return the number of packets to be processed by the pattern, and getStageCount method should return the total number of stages needed for the job. The other skeleton that has been implemented is the workpool. The patterns that have been imple-

mented include the 5-point stencil and the all-to-all pattern. The implementation for these patterns has been developed using many of the parallel programming techniques mentioned in the advanced layer section. However, the basic user does not need to be aware of the technique used in the lower layer, since (ideally) the skeleton/pattern interface should stay the same. For example, the interface shown in Figure 2.10 can be used to run the pipeline skeleton using a data flow implementation, or an ordered template implementation.

```
public abstract class PipeLine extends BasicLayerInterface {
      public abstract  Data Compute(int stage, Data input);
      public abstract Data DiffuseData(int segment);
      public abstract void GatherData(int segment, Data dat);
      public abstract int getDataCount();
      public abstract int getStageCount();
}
```

FIGURE 2.10: Example of a pipeline interface.

2.9 Testing and Experimentation Equipment

Most of the experiments done with the framework, and its extensions, include a performance elements. The general method used to test the performance is to measure the time it takes for a specific, well defined, task to complete. The measurement is taken several times. The result presented is then the average of several runs. The use of multiple runs smooths fluctuations in performance due to other characteristics inherent in the operation system. The relative standard error may be shown to provide additional information about a benchmark or validation experiment. The result can also be expressed as speedup [51]. Speedup is defined as:

$$Speedup = \frac{S}{P} \tag{2.1}$$

In Equation 2.1, *S* is the time taken for the task using a single processor. *P* is the time taken to complete the task using multiple processors. The speedup for a single processors is 1. The ideal speedup is define as:

$$\frac{S}{P(n)} = n \tag{2.2}$$

From Equation 2.2, ideally, the task should be completed faster by a multiple of *n* when using *n* more processors when compared with the time taken by a single processor.

Because the performance of the system is important in these measurements, we will mostly refer the two systems shown on Table 2.1 when discussing results. "Coit-grid Shared Memory" was used to test the framework in a multi-core environment and the "Coit-grid Cluster" was used to test it on a distributed memory environment.

TABLE 2.1: A cluster and a multi-core server were used for multiple tests.

	Coit-grid Cluster	Coit-grid Shared Memory
Number of Computers	4	1
Total Memory/server	8GB	64GB
CPUs/server	2	4
Threads/server	4	16
CPU Speed	3.4GHz	2.93GHz
Network Type	Gigabit	Gigabit
Operating System	Red Hat Linux	Red Hat Linux
Kernel version	2.6.9-42 and 2.6.18-92	2.6.18-164
Java JVM version	HotSpot(TM) 64-bit Server 10.0-b22 mixed mode	

The other tool used to evaluate the results is counting the lines of code (LOC's). This measurement is used to assess the concept's usefulness in terms of programmability. In general, a better result can be seen as a program that uses less LOC's and still parallelizes

the algorithm. It should be noted the LOC's targeted in this research are the non-functional lines of code. We first separate the LOC's into functional, non-functional, and automatic:

- Functional: The code that is dedicated to solving the problem. Most of the LOC's in the a serial implementation are considered functional.
- Non-functional: The code primarily written to organize parallel processes and communications. MPJ-Express, and Seeds include code of this type to solve the problem in parallel.
- Automatic: This code is generated code by the IDE. Eclipse was used for the test. The generated code includes the class declaration, import lines, package declaration, and interface signature functions. Setters and getters are also included as automatic code.

With the three types of LOC's defined, we can use the programmability index, which is shown on Equation 2.3.

$$programability\ index = \frac{functional}{functional + non\text{-}functional} \tag{2.3}$$

The programmability index goes from zero to one, with one being a program that can parallelized using no non-functional LOCs, and a near zero index means a program that needs many LOC's to achieve parallelism.

2.10 Seeds Speedup Benchmark

The Seeds framework has been compared to MPICH2 [52] and MPJ Express [53] to assess its performance. It is expected the framework will approach MPJ Express performance once optimized. The tests were performed on the Coit-grid Cluster. The bench-

mark algorithm was the approximation of π using a Monte Carlo approach. The problem was implemented using a workpool on Seeds, and using standard MPI directives in MPJ-Express and MPICH2. Figure 2.11 shows the speedup for each platform. To produce these results, the program was run on a single core with compiled serial C++ code, and again in Java. The serial time was divided by the total time of each run to get the speedup.

One reason for the inefficiencies is the overhead incurred by using objects as a vessel for the data type. The framework has not been thoroughly scanned for potential resource overuse as can be done by profiling the framework. Techniques to reduce the overhead due to object serialization are presented in Chapter 5. Also, better performing communication protocols can be used with the framework such as Java FastSockets[54], a native interface that increases network performance for Java applications on Gigabit Ethernet

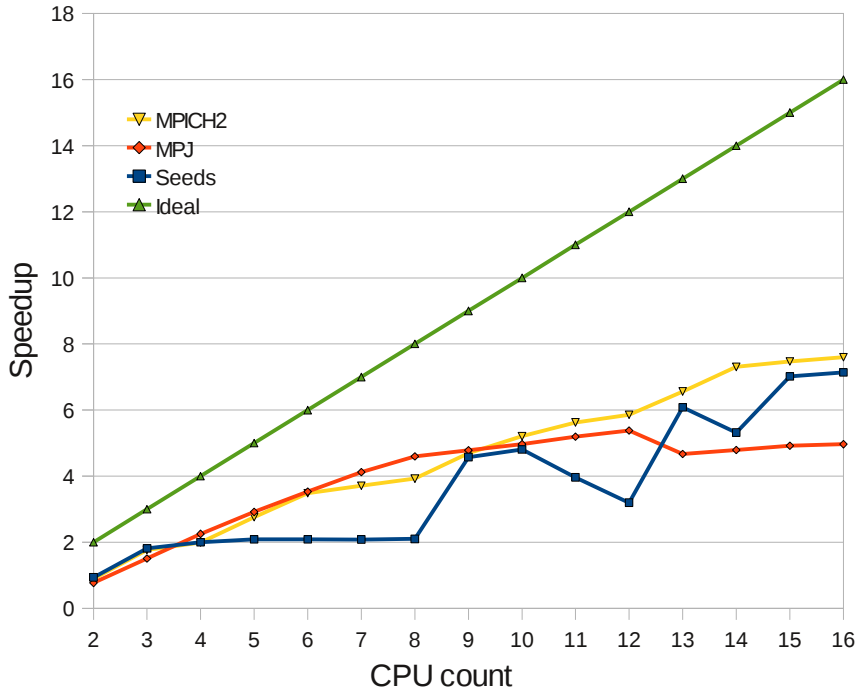


FIGURE 2.11: Speedup for the frameworks: MPICH2, MPJ Express, and Seeds.

and other network standards. MPI uses customized protocols to reduce overhead, and TCP is used for the Java framework socket.

2.11 Conclusions

Seeds has been very useful in studying and extending parallel programming concepts into the Grid and heterogeneous environments. Although some dead-ends were encountered in the journey towards creating new solutions for the more complex parallel environments of the future, overall the framework has been useful to test research ideas. This chapter presented the overview of what requirements were implemented into Seeds, and why there was a need for this new framework. A review of the JXTA platform was presented since JXTA is extensively used by the Seeds framework. The report presented a detail explanation on how the communication lines are abstracted in Seeds, and how the processes are managed to handle parallel programming patterns as well as other parallel programming techniques. The last section presented a comparison in performance among Seeds and other parallel programing techniques.

CHAPTER 3: PATTERN OPERATORS

Pattern operators are extensions to the pattern/skeleton parallel programming approach used to apply two types of communication patterns to the same data. The operators are proposed to simplify the wide range of possible patterns and skeletons into patterns that can be closer to a basic set. The basic set of patterns, the operators and other features such as nesting of skeletons, can be used to create all possible parallel algorithms. This can relegate MPI/OpenMP to a lower layer of abstraction, and provide the Grid/Cloud environment with needed communication abstractions. The abstraction helps manage non-functional concerns in these environments. This chapter explains how the pattern operators work on synchronous cyclic undirected graph patterns, and it shows examples on how they are used. A prototype was created to test the feasibility of the idea. The example used to show the operator approach is the addition of termination detection to a discrete solution to a Partial Derivative Equation (PDE). The example can be coded with 27.31% less non-functional code than a similar implementation in MPJ, and its programmability index is 13.5% compared to MPJ's 9.85%. The overhead for an empty pattern with low communication was 15%. The use of pattern operators can reduce the number of skeletons/patterns developed, thereby reducing the probability that a user programmer will need to develop new patterns when using this programming approach.

Grid and cloud computing require the use of abstractions such as skeletons/patterns

(SPs), particularly, the need to separate the resources of the cloud from the users and developers. Section 1.1 discussed the advantages and the challenges associated with skeletons and patterns. This chapter presents research on reducing the need to create more complex patterns when a combination of simpler patterns can be used instead.

When developing an application, the programmer may be faced with a library of skeleton/patterns. The list can grow to be large as skeleton and pattern developers work to create new patterns to fulfill some particular job. The downside to this is that a large list can discourage development using the skeleton/patterns because the programmer can become overwhelmed by the choice. On the other extreme, the programmer, despite having multiple skeleton/patterns to choose from, does not find the pattern that is needed for the problem, and therefore is tempted to just use lower level tools.

One can intuitively surmise that there can be a basic set of skeleton/patterns from which all the other skeleton/patterns can be created by adding extensions to the skeleton/patterns concept. One such extension is nesting skeletons. In nesting, the user programmer is able to deploy new skeleton/patterns from inside the skeleton/patterns, which allows for an exponential increase of skeleton/patterns without increasing the size of the basic set. Nesting also allows for the use of libraries that contain skeleton/patterns themselves. With some operators and a basic set, one could see a Turing-complete (so to speak) set of patterns from which all possible parallel programs can be created.

The effort behind the pattern adder operator is to reduce the number of custom patterns that have to be created in order to develop algorithms with complex communication patterns. However, the technique is not expected to provide a complete basic set of pat-

terns from which all parallel programs can be created. Providing a basic set is difficult because there are algorithms that use many different communication patterns either as a result of the domain specific algorithm, or as a result of research on how to optimize the implementation of the algorithm for high performance. An example of this is the wave-front pattern. This pattern is developed similarly to a 5-point stencil, but the communication pattern will propagate messages as in a pipeline from top left, top, and left cells to the local cell. This simple example shows that there can always be an exception to the creation of a basic set of patterns. Other examples include algorithms to process dense matrices.

The issue of having multiple communication patterns for which no parallel pattern exists has been encountered by previous work such as CO_2P_3S[14]. The solution implemented by those projects, which is also implemented into Seeds, is to create an extra layer of development that we call the advanced layer, so that an advanced programmer can create new patterns if the pattern sought after does not exists.

The pattern adder operator, therefore, looks to reduce the number of instances where an advanced user needs to create a new pattern by breaking up the patterns into multiple layers of communication patterns. This gets us closer to a basic set while leaving room for the creation of less common communication patterns.

We will begin first with the introduction of nested skeletons and their implementation for the Seeds framework on Section 3.1. Next, Section 3.2 presents a high level explanation about pattern operators. Subsection 3.4 presents an example using Java. It explains the use of interfaces to create computation modules and the interfaces used to create data

containers. Subsection 3.5 presents the implementation of the adder operator in Seeds. Section 3.6 presents the results from measuring the adder operator on the dimensions of performance and programmability. Finally, Section 3.7 mentions the related work specific to the pattern operator concept.

3.1 Nested Skeleton

Nesting skeletons consist of providing the API's and infrastructure to allow the domain specific programmer to deploy new skeletons within the main computation method for a currently running skeleton. The nesting effect pauses the computation on the current skeleton, deploys the new skeleton, and waits for the algorithm to finish. Once the nested skeleton is finished, control flow returns to the parent skeleton, and it can continue until completion.

Nesting skeletons have two advantages. First, they can be used to create more complex skeletons out of simple skeletons. Figure 3.1 shows an example. In this example the user is creating a divide-and-conquer algorithm using workpool skeletons. The diffuse method is used to perform the divide part, and the compute method is used to either

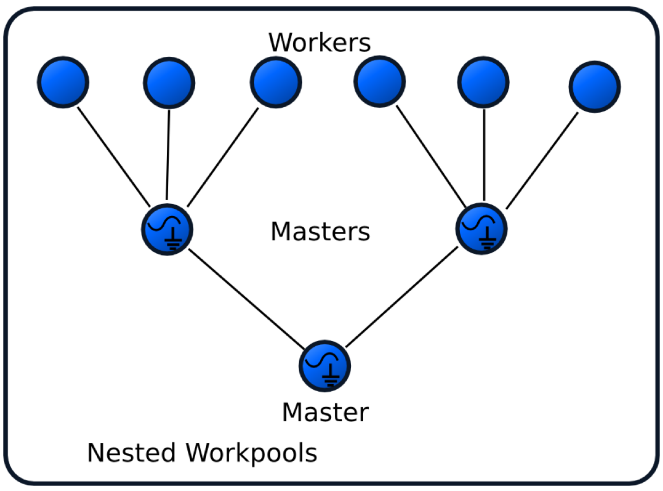


FIGURE 3.1: Nested workpool used to create a divide-and-conquer skeleton.

deploy a new workpool pastern or compute the conquered data.

Second, the nesting feature allows for the creation of libraries that use the parallel framework's routines within it. Figure 3.2 shows an example of this benefit. In the example, a user needs to sort a list of objects. To accomplish this, the source code calls a sorting library. Unknown to the programmer, the sorting library itself uses a skeleton to sort the list taking advantage of the parallel environment. In the example, the framework allocates the processes based on the system's load.

One of the drawbacks to nesting is that deploying a new skeleton has more overhead when compared to a parallel application that arranges the processes to work together from the beginning [9]. Aldinucci et al. created a label transition system that is used to symbolically manipulate and simplify (if possible) skeletons to reduce the number of nesting that may need to happen.

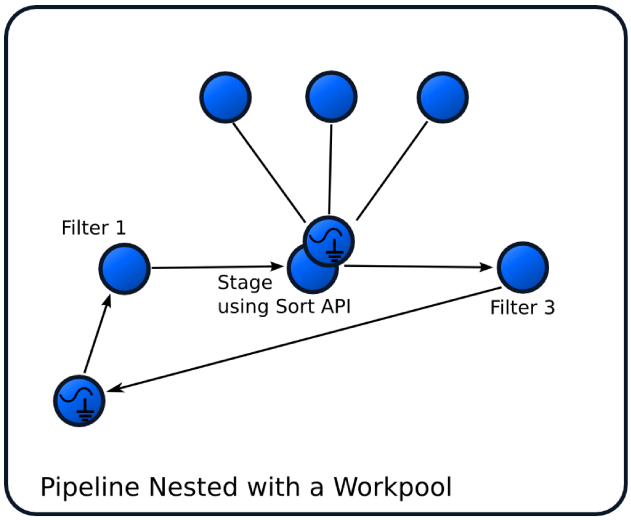


FIGURE 3.2: A pipeline with a nested workpool.

3.2 Pattern Operators

The creation of a pattern adder operator comes about to address stateful algorithms such as discrete solutions to PDE's and practical solutions to particle dynamics algo-

rithms. In these types of algorithms, there are different communication patterns at different stages of programming. In the example of a discrete simulation of heat distribution, multiple cells on a stencil pattern work in a loop parallel fashion, computing and synchronizing on each iteration. However, every x iterations, they must implement an all-to-all communication pattern to run an algorithm to detect termination. That is used to check if all cells have converged on a value and all the cells should at that point stop computing. Figure 3.3 shows the example of this approach.

It is easy to see that many more algorithms fall into this category where one has multiple layers of communication patterns that work on the same data. Like nesting, the multiple layers of algorithms may not be coded at the same time, they may instead be provided as libraries. Therefore, the programmer can benefit from a tool that can add the "filter" to the toolkit without having to re-implement an existing algorithm. In the example of heat distribution, the data is a set of pixels that represent the heat energy present at that point. In the case of particle dynamics, the data represents momentum for each particle at that instant in time.

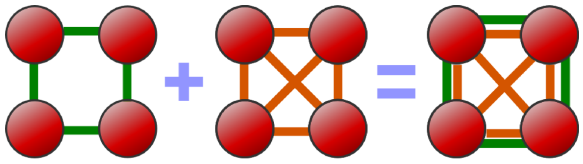


FIGURE 3.3: Adding a Stencil plus an All-to-All synchronous pattern

3.3 The Heat Distribution Problem

The heat distribution problem was used to test the patter operators by adding a termination detection algorithm to it, and was also used to test the 5-point stencil implementation used to test hierarchical dependencies in Chapter 4. Another applicable algorithm

for this type of pattern is the the multi-grid algorithm[55]. Other algorithms that show synchronous communication behavior exist. However, their implementation resembles the one done with the heat distribution problem, with only the main computation changing.

The heat distribution algorithm is a discrete solution to the steady state of a partial differential equation [56]. The equation is of the form:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0 \tag{3.1}$$

Using discretization techniques, the formula can be represented in a more computer friendly representation

$$u_{j,l}=\frac{u_{j+1,l}+u_{j-1,l}+u_{j,l+1}+u_{j,l-1}}{4} \tag{3.2}$$

Finally, using Jacobi iteration, the heat distribution equation can be calculated by using the previous values from the neighbor points of j,l

$$u_{m,l}^{(m+1)}=\frac{1}{4}\left(u_{j+1,l}^{(m)}+u_{j-1,l}^{(m)}+u_{j,l+1}^{(m)}+u_{j,l-1}^{(m)}\right) \tag{3.3}$$

We keep two matrices, one with the old values $u^{(m)}$ and one with the new values $u^{(m+1)}$. Figure 3.4 shows this part of the algorithm.

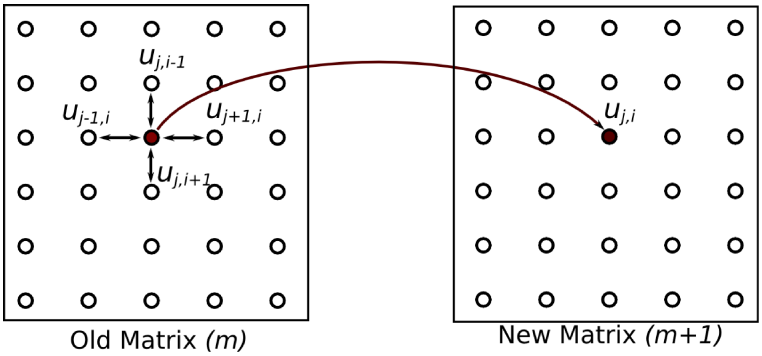


FIGURE 3.4: Computing one point from a 2D matrix using the heat distribution formula.

The serial version for the problem is shown on Figure 3.5. The source code is of importance because it provides lines of code (LOC's) to measure programmability in the implementations. The LOCs are tagged on the implementation with the tags: functional, non-functional, automatic, comment, and log.

The programmability index is computed as shown on Section 2.9. A program with a higher programmability index has less non-functional code, and therefore is more readable and easier to parallelized than one with more non-functional code. For example, the heat distribution implementation shown in Figure 3.5 has 31 lines of functional code, sev-

```
1. public class HeatDistributionSerial {
2.     public static void main(String[] args) {
3.         double[][] matrix = MatrixReader.getMainMatrix();
4.         MatrixReader.saveImage(matrix, "saved1.png");
5.         boolean terminated = false;
6.         while( !terminated ){
7.           double[][] m = new double[matrix.length][matrix[0].length];
8.           for( int i = 0; i < matrix.length ; i++){
9.                m[i][0] = matrix[i][0];
10.               m[i][matrix.length - 1] = matrix[i][matrix.length - 1];
11.               m[0][i] = matrix[0][i];
12.               m[matrix.length -1][i] = matrix[matrix.length -1][i];
13.          }
14.          for( int r = 1; r < matrix.length -1; r++){
15.               for( int c = 1; c < matrix[0].length -1; c++){
16.                     m[c][r] =     0.25 * (  matrix[c + 1][r] +
17.                                             matrix[c - 1][r] +
18.                                             matrix[c][r + 1] +
19.                                             matrix[c][r - 1]  );
20.               }
21.          }
22.          double max_diff = 0.0;
23.          for( int r = 0; r < matrix.length  ; r++){
24.               for( int c = 0; c < matrix[0].length ; c++){
25.                     double diff = m[c][r] - matrix[c][r];
26.                     if( diff > max_diff) {
27.                           max_diff = diff;
28.                     }
29.               }
30.          }
31.          terminated = max_diff < 10.00;
32.          matrix = m;
33.        }
34.        MatrixReader.saveImage(matrix, "saved.png");
35.   }
36. }
```

FIGURE 3.5: Serial implementation for the heat distribution problem.

en lines of automatic code, and zero non-functinal lines of code. The serial implementation does not have nonfunctional lines of code, therefore it is a good statistical control to be used to measure other programming approaches. Computing the programmability index for the serial implementation produces a 100% programmability index.

3.4 Implementing the Heat Distribution Problem using Pattern Operators

Figures 3.6 shows how the heat distribution problem is implemented using a 5-point

```
1.  public class HeatDistribution extends Stencil {
2.   private static final long serialVersionUID = 1L;
3.   int LoopCount ;
4.   public HeatDistribution(){
5.    LoopCount = 0;
6.   }
7.   @Override
8.   public StencilData DiffuseData(int segment) {
9.    int w = 10;
10.   int h = 10;
11.   double[][] m = new double[10][10];
12.   /** init matrix m with file or user input*/
13.   HeatDistributionData heat = new
14.                HeatDistributionData(m, w, h);
15.   return heat;
16.  }
17.  @Override
18.  public void GatherData(int segment, StencilData dat) {
19.   HeatDistributionData heat = (HeatDistributionData) dat;
20.     /** print or store results*/
21.  }
22.  @Override
23.  public boolean OneIterationCompute(StencilData data) {
24.   HeatDistributionData heat =
25.
                (HeatDistributionData) data;
26.   double[][] m = new double[heat.Width][heat.Height];
27.   /** compute core matrix */
28.   /** compute sides (borders)*/
29.   /** compute corners */
30.   /** set if this node is done*/
31.   heat.matrix = m;
32.   return false;
33.  }
34.  @Override
35.  public int getCellCount() {
36.   return 4; //four nodes for this example
37.  }
38.  @Override
39.  public void initializeModule(String[] args) {
40.   //not used
41.  }
42. }
```

FIGURE 3.6: HeatDistribution class extends Stencil and fills in the required interfaces.

stencil pattern on the Seeds framework. Some of the problem-specific code was omitted in the interest of brevity. The class HeatDistribution extends a Stencil abstract class. This requires the programmer to implement some signature methods. The Javadoc for each signature method is used to instruct the programmer on the purpose of each method and their interaction within the framework.

- The DiffuseData(): The method is used to get the segments of data from the user programmer. For our example, this is a 2D matrix which is a submatrix for the larger matrix. If the user requests 4 processes to compute the pattern, the Diffuse-Data() method is called 4 times with the segment ID going from 0 to 3.
- GatherData(): is used to get the processed segments of data back from the user. In this method, the programmer is expected to join the sub-matrices back together to form the initial matrix, by this time containing the final answer.
- OneIterationCompute(): is used as the main computation method. Because the algorithms are loop-parallel and the framework needs to gain back control in order to organized multiple patterns, the user is instructed the method should only run one iteration of the main loop in the application. The user is given the stateful data and the chance to compute one iteration. Once the computation for the iteration is done, the method is done returning *false*. If the method returns *true*, this signals to the framework the program is done.
- InitializeModule(): is used to allow the user programmer to pass string arguments to the remote instantiation just after the modules get initialized.

For distributed memory environments, another algorithm must be used to synchronize

termination for the parallel application. One way to solve this is to run a constant number of iterations before exiting. Another approach is to include an algorithm that measures the answer computed so far. For the heat distribution example, the tolerance measures (lines 22 to 30 in Figure 3.5) how close the piece of material is to homeostasis. Measuring a condition on which to stop can help reduce computation time by avoiding computations that marginally add to the final answer. However, the use of termination detection on a parallel program depends on the program implemented and the answer desired.

If needed, pattern operators can facilitate the process of adding a termination detection algorithm to a synchronous program. Figure 3.7 shows an example where an all-to-all termination detection algorithm is used to determine if there is convergence after performing a stencil algorithm for some number of iterations.

Figure 3.7 shows the TerminationDetection class, which extends CompleteSyncGraph

```
public class TerminationDetection extends CompleteSyncGraph {
 @Override
 public AllToAllData DiffuseData(int segment) {
  // not used
  return null;
 }
 @Override
 public void GatherData(int segment, AllToAllData data) {
  // not used
 }
 @Override
 public boolean OneIterationCompute(AllToAllData data) {
  HeatDistributionData d = (HeatDistributionData) data;
  return d.Terminated;
 }
 @Override
 public int getCellCount() {
  // not used really
  return 4;
 }
 @Override
 public void initializeModule(String[] args) {
  // not used.
 }
}
```

FIGURE 3.7: Termination detection using all-to-tall pattern.

class. CompleteSyncGraph is the interface used to implement an all-to-all pattern. Similar to the stencil pattern, CompleteSyncGraph also requires some signature methods. The pattern has DiffuseData() and GatherData() methods but they are not used for this example since the second pattern in the operator is used for its computation function only. If the all-to-all pattern was to be used by itself (not as part of an adder operator) DiffuseData() and GatherData() would need to be implemented by the programmer. getCellCount() is the number of processes needed for the computation and must return the same number on both patterns so that communication patterns fit together.

Figure 3.8 shows the main data object used for both the patterns. The main advantages sought in using the pattern adder is to provide the user programmer with the ability to have two communication patterns work on the same data. Our approach to patterns has the requirement of having all information used for communication travel in the form of serializable objects. Additionally, the stencil pattern adds other signature methods that are needed in order to control the communication on behalf of the user programmer. From line 10 to 26 are signature methods for the stencil pattern. The methods are used by the framework to retrieve data from the user's module, and to store data into the user's module visible memory. The data the framework moves in and out of the programmer's module is the data exchanged with the neighbor cells (from remote nodes). The timing of the calls for these signature methods are communicated to the programmer through documentation, but the implementation for the calls are hidden for the programmer convenience. CompleSyncGraph also adds signature methods in lines 27 to 35. The data retrieved from the module using getSyncData() method is sent to all the other processes,

```
1.  public class HeatDistributionData
2.        implements StencilData, AllToAllData {
3.   boolean Terminated;
4.   public double [][] matrix;
5.   public int Width,Height;
6.   SyncData[] Sides;
7.   public HeatDistributionData(double [][]m
8.             , int width, int height){
9.   }
10.  /** Stencil data signature methods */
11.  @Override
12.  public Data getBottom() {}
13.  @Override
14.  public Data getLeft() {}
15.  @Override
16.  public Data getRight() {}
17.  @Override
18.  public Data getTop() {}
19.  @Override
20.  public void setBottom(Data data) {}
21.  @Override
22.  public void setLeft(Data data) {}
23.  @Override
24.  public void setRight(Data data) {}
25.  @Override
26.  public void setTop(Data data) {    }
27.  /** The All-to-All Data signature methods */
28.  @Override
29.  public Data getSyncData() {
30.   /**return data for all*/
31.  }
32.  @Override
33.  public void setSyncDataList(List<Data> dat) {
34.   /** get data from all */
35.  }
36. }
```

FIGURE 3.8: The main data extends both the StencilData and AllToAllData. The object is used to hold the state-full data for the main processing loops.

and the data gathered from the other processes are provided to the programmer in list form using setSynchDataList() method. HeatDistributionData implements both StencilData and AllToAllData so that it can be handled by both the stencil pattern and the all-to-all pattern.

Both of these modules are inserted into the framework using a bootstrapping executable class. Figure 3.9 shows the executable the user programmer implements in order to add the stencil pattern plus the all-to-all pattern (lines 7 through 22) . The two are

```
1.  public class RunHeatDistribution {
2.   public static void main(String[] args) {
3.    Deployer deploy;
4.    try {
5.     Seeds.start("/path/of/shuttle/folder/pgaf", false);
6.      /**first pattern */
7.      Operand f = new Operand(
8.          (String) null
9.          , new Anchor("Kronos"
10.                , DataFlowRoll.SINK_SOURCE)
11.         , new HeatDistribution() );
12.     /**second pattern*/
13.     Operand s = new Operand(
14.         (String) null
15.         , new Anchor("Kronos"
16.               , DataFlowRoll.SINK_SOURCE)
17.         , new TerminationDetection() );
18.     /**create the operator*/
19.     AdderOperator add =
20.          new AdderOperator(
21.          new ModuleAdder( 100, f, 1, s )
22.          );
23.     /**start pattern and get tracking id*/
24.     PipeID p_id = Seeds.startPattern( add );
25.     /**wait for pattern to finish*/
26.     Seeds.waitOnPattern(p_id);
27.    Seeds.stop();
28.   } catch (Exception e) {
29.    /**catch exceptions*/
30.   }
31.  }
32. }
```

FIGURE 3.9: RunHeatDistribution is used to create the operator and start the pattern.

added using an Operand class that is used to hold together three characteristics each pattern needs, which are:

- Initialization Arguments: these string arguments are sent to the remote nodes and given back to the programmer using the method initializeModule().
- Anchors: the anchor is the host used to feed information to the other remote nodes. Most often, the anchor tends to be the host from which the parallel program is launched.
- Pattern Module: An instance of the pattern module. For the heat distribution example, this would be an instance of HeatDistribution class.

The executable class also has some code to start the framework and shutdown the framework (lines 5 and 27). After creating the operands, the pattern-adder operator is deployed by starting a new pattern called an AdderOperator (line 19). The framework, by default, will spawn and monitor the new pattern on a separate thread. The programmer's launching thread can wait for the pattern to complete using waitOnPattern() method.

3.5 The Operators

The operators at present are only implemented for the OrderedTemplate pattern because only the addition operators seems to be beneficial in reducing pattern complexity for the user programmer. Future endeavors may include adding operators to unstructured skeletons, to get similar benefits as we show can be had from the addition operator.

The operators are implemented by inheriting the OrderedTemplate class. Once this OrderedTemplate is loaded by the framework and readied to execute, the OperatorTemplate runs the first pattern to load the initial computational units (by calling DiffuseData() method). Then it enters into the main loop-parallel cycle where the OneIterationCompute() methods from the module objects will be called. It runs the first operand from the first module for *n* iterations, and then it runs the next pattern for *x* iterations. The process is repeated until either one of the patterns return true. The computation for these patterns return true if the program is done computing. When the main loop-parallel cycle is done, the operator pattern returns the processed data units to the first operand by calling GatherData() method.

The operator implements LoaderTemplate to load and unload the initial data from the first pattern. The second pattern only contributes the computation operation, and the Diffuse/Gather operations are ignored by the operator executor. Figure 3.10 shows a diagram that describes most of the interaction among the classes that happens when running the operator template. The small tabs inside the square are used to mention the BasicLayerInterface class that is used by the Template class. For example: Stencil class inherits BasiclayerInterface, and it is used by StecilTemplate class. Together, both Stencil and StencilTemplate implement a stencil pattern. The two patterns are added into the PatternAdder interface that is executed by the AdderTemplate. Because the adder template is an OrderedTemplate, it must specify a PatternLoader interface, and that is done by the PatternAdderLoader class. PatterAdderLoader inherits PatternLoader interface, and it is executed by the PatternLoaderTemplate. Finally, Seeds can execute the LoaderTemplate direct-

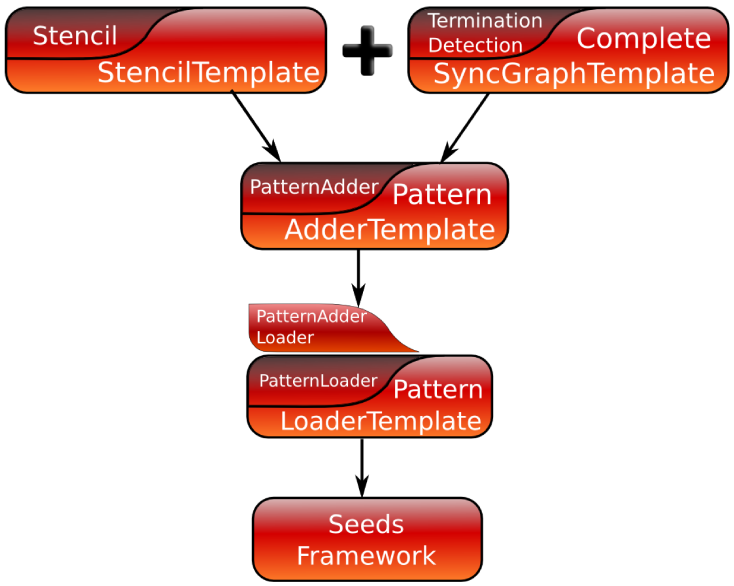


FIGURE 3.10: Interface and Template pairs are drawn on the same square. The diagram shows the hierarchical interaction between the classes in order to execute a PatternAdder operator.

ly because it is an UnorderedTemplate. All the system of modules just explained is loaded into the virtual machine using Java reflection libraries. The reflection library allows the software to load a class for which the framework may not know the qualified name.

The end results is a simplified programming environment for the programmer. Most of the complexity added to this implementation hovers around having to deal with software that does not exist since the user programmer has not developed it yet. In the face of that, the presented design allows the framework to implement the pattern operator on general patterns despite not knowing the specific program.

All templates implement a function for the client side node and one function for the server side node. The server side corresponds to the source and sink nodes, and the client side corresponds to the compute nodes.

3.6 Results

Tests were performed to validate the pattern adder operator. The two main concerns for extensions to the pattern programming approach are the performance impact created by the extension, and programmability of the extension.

On the performance side, the performance overhead was of interest. The performance overhead is the extra time incurred by the framework to add the two patterns together in comparison to how long it takes the framework to run each of the patterns by itself. In the test, the stencil pattern and the all-to-all pattern are run independently. The time from the two independent tests are added to make the ideal, overhead-free time. Then, the two patterns are added using the pattern adder operator. The difference between the latter and the former test is considered the overhead.

In order to measure the performance overhead created by the pattern adder operator, we implemented a simple algorithm that uses both the stencil pattern and the complete pattern. The algorithm is trivial; it consists of sending a long integer type to the neighbor processes in the stencil pattern, and it repeats the process for the complete pattern. We consider this an *empty grain size pattern*. The time to run through one iteration is measured for the stencil pattern and for the complete pattern. The time taken to run the process is also measured for the pattern adder operator. The overhead is the difference between the pattern operator's time and the stencil plus the complete pattern's time. Figure 3.11 shows the result of this test. All the communications for this experiment were through shared memory using the Coit-grid Shared Memory system. The results show that the overhead goes down as more processes are used for the computation. This is in part because the increasing communication overhead helps mask the overhead due to the operator. The overhead in comparison to an empty grain size is 15%, so grain size has to be adjusted to justify the use of the operator. The network speed also has an effect on the

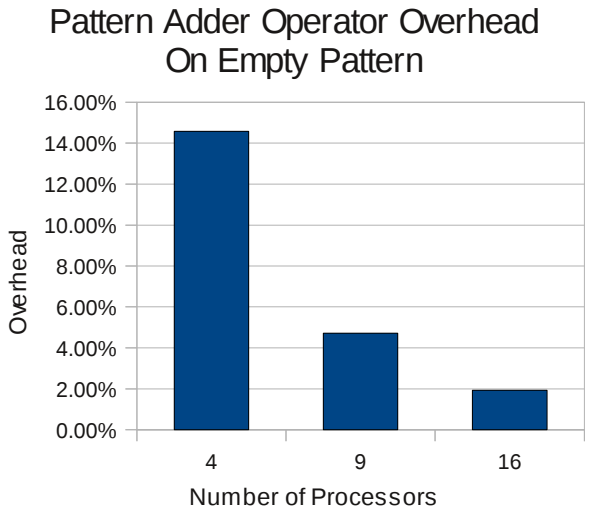


FIGURE 3.11: Operator overhead measured on a shared-memory multi-core server.

overhead. As Figure 3.11 shows, the increase in communication overhead reduces the overhead incurred due to the operator. The same test was performed on the Coit-grid Cluster. The overhead for this test on an empty grain size pattern was 0.03% for nine processes. The network used was a Gigabit Ethernet.

Next, we measure the programmability. For this test we implemented the heat distribution algorithm using MPJ-Express. We also implemented the problem using the Seeds framework, and a serial version of the problem was used as control. For all three implementation, the task of loading and unloading the matrix from a text file was done using a separate matrix manipulation class. Since all three implementation use the class, we do not count the lines from that class on any of them.

Figure 3.12 shows the result from this assessment. Seeds reduces the number of non-functional code by 27.31% over MPJ implementation. MPJ's programmability index for this implementation is 9.85%, and Seed's programmability index is 13.50%.

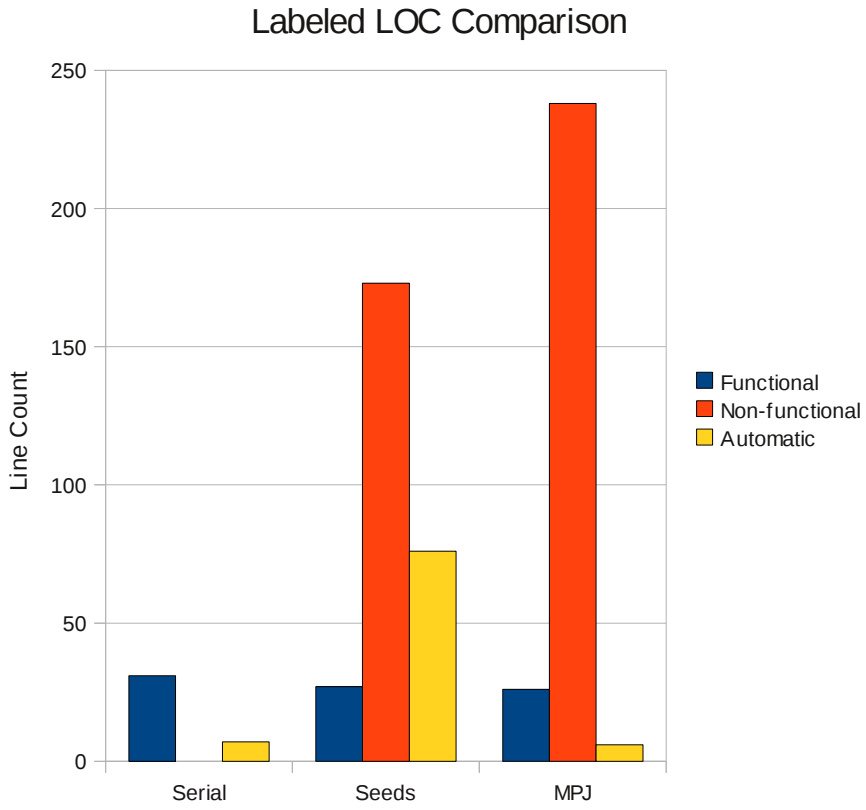


FIGURE 3.12: The y axis shows the number of lines of code for each implementation. The LOC were counted for the serial implementation as well as for the Seeds and MPJ implementation.

3.7 Related Work

The nested feature is important for any skeleton/pattern framework. Lithium [9], Muskel [57] and other frameworks have implemented the feature [15]. The use of pattern operators comes closest to the work of Gomez et al. Their pattern operators implement the same concept that set us in the direction to create the pattern operators [58]. Their work is based on workflows and includes other types of operators that are used to manage non-functional concerns. The programmer in the Triana framework is given more control over the resources where the program runs. Our work differs from Triana in that we provide the pattern operators as a pure object-oriented framework without the need for XML

language constructs, scripts, or a GUI. Also, we have measured the effects of the tool. The use we intend for pattern operators is targeted toward high performance parallel computing in a heterogeneous environment.

3.8 Conclusions

We proposed an object oriented implementation to provide an adder operator to skeletons/pattern parallel applications. A sample program was shown and its creation was discussed from the user programmer's perspective. Subsequently, the implementation of the pattern adders was presented. The advanced user's perspective was discussed, and some notes about the Seeds framework and the expert programmer's perspective was also discussed. The pattern operator can be used to reduce the number of patterns that are needed by the user programmer. We believe that providing a basic set of skeletons/patterns plus useful operators will increase the popularity of this parallel programming model.

CHAPTER 4: HIERARCHICAL DEPENDENCIES

Chapter 3 dealt with problems in making skeleton/patterns more accessible. However, one of the main objectives for this research has been to create an abstraction layer that separates the programmer from the resources being used. In turn, the layer should allow framework experts to address many non-functional concerns. In this chapter, we tackle a couple of non-functional concerns: grain-size and scalability. For this, we first imaging a finished, production-level, framework with many other required modules that were not developed for Seeds (refer to Section 1.4). The finished framework should be able to load balance and schedule a parallel program without having a priory knowledge of the specific algorithm. It should just have knowledge about its skeleton/pattern. In Chapter 3, we already introduced the concept of returning execution flow to the framework after each iteration from the main loop (As noted before, we assume that most computation and data intensive applications have a main loop.) In this chapter, we use that concept, in conjunction with an algorithm to stop computation, and the use of data flow to facilitate automatic scalability into synchronous patterns. The automatic scalability also implies a grain-size reduction for most synchronous parallel programs. The concept is called hierarchical dependencies for a reason explained later in the chapter.

Hierarchical dependencies' main goal is to provide an interface to the programmer that can be used to automatically split data size, and split the amount of computation based on

a changing environment of CPU resources. The environments we predict will take advantage of a dynamically changing number of computation resources include the Grid computing field, the cloud, and multi-core computing.

The problem that exists currently is that parallel applications must request the constant number of processes needed before starting to compute a solution. This is fine if the program is created for a static cluster, but not so if the program will run on a heterogeneous environment. Still, there are other programs that can be adapted to run on different configuration of processes, but even this is coded by the basic programmer. The issue with this is that the domain-specific programmer may not be an expert in parallel programming, and there are many technical challenges that can occupy a basic programmer's time who is working outside his field of expertise.

The advantages in changing the number of processes dynamically are the ability to control grain-size and to optimize performance. This is possible because distributing a job among more processors should also reduce the time to completion as long as the grain-size does not get too small. If the grain-size is too small when the application starts computing, reducing the number of processors can help optimize performance by coalescing processes and data, thereby reducing the overhead of running on extra processes. Coalescing the processes is more effective than simply allocating multiple processes on a singe processor because it reduces the overhead of running the parallel application. The overhead includes increase demand on the OS to manage more threads, and unnecessary synchronization points for processes working on the same processors.

Being able to dynamically manage the number of processes can also be of use for mul-

ti-core machines. Some problems can have varying degrees of work. For example, interactive applications can go from periods of idling to periods of heavy computation depending on the tasks done by the user. With these changes, a desktop application can expand and contract on the number of cores being used depending on the program. This makes the parallel application more efficient, and it can allow the operating system to either run another application in the idle cores, or to shutdown the unused cores to save energy.

Lastly, the automatic adaptability can be used to redesign schedulers and load balancers. A load balancer can create performance profiles that can be used to shift the work load from one processor to another. With hierarchical dependencies, the load balancer can also decide if the application can work more optimally if the number of processes are increased or decreased. If the conditions are met, the load balancer can direct a clique of perceptrons to split. At that point the perceptrons will hibernate and return to the master node to be split. Upon splitting, the master node will redeploy the child perceptron. The child perceptron then incorporates new hierarchical dependency IDs that are compatible with their parent dependencies. The hierarchical syntax is used to find and connect child dependencies with other child dependencies, or to connect parent dependencies with child dependencies. The end result is an application that can increase its scalability with little or no effort on the side of the domain-specific programmer. This can be done because using the auto-scalability, a program is not as static as assumed to be by current literature on schedulers and load balancers. This adds another parameter to the load balancer where the application can be directed to shrink in resources or expand.

A scheduler could also be modified to manipulate existing applications to make the necessary resources available to launch a new application. This suggests a scheduler and load balancer can be better designed by creating a module that does both tasks. Research into schedulers and load balancers is left as future work as noted in Section 1.4.

Having a framework that allows the parallel program to change according to the environment's conditions can allow the application to perform optimally. A research question we have to answer is how much overhead would such a solution add ? Also, we measure how much extra effort it takes to add the concept to an application.

The next section reviews the concept behind data flow parallel programming. It presents technical details specific to the Seeds framework, and how the framework's API should be used by an advanced and basic programmers to create patterns and problem solving programs respectively. Section 4.4 introduces the idea of hierarchical dependencies. It explains the main extension provided by these dependencies over the approach implemented by previous work. The hierarchical dependency in Section 4.11 concept is divided into: the dependency identification syntax, how dependency split procedures are implemented, and how the hibernation procedure is done. Section 4.12 presents the experiments done to test the concept. Finally, Section 4.14 reviews related work.

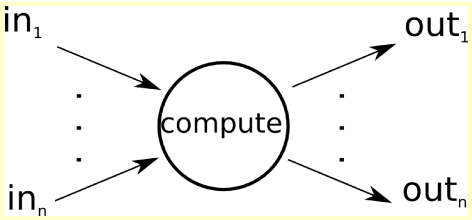
4.1 Data Flow Model

A data flow is a directed acyclic graph where the vertices represent processes, and the edges represent communication lines between the processes. Each of the processes therefore can be represented as a unit with a set of inputs, a set of output, a state, and a computation that is done on its inputs to create the outputs. This section presents the implementation of such a parallel programming structure onto the Seeds framework and how the

data flow implementation is attached to the existing framework's lower layers. The implementation can be divided into: data flow unit implementation, interfaces used by the advanced user, and interfaces used by the basic user.

In Seeds, the data flow nodes are Java objects that can be sent over the network to the remote processes. The term perceptron will be used to refer to an individual data flow node within the data flow network. The term is borrowed from neural network nomenclature since a data flow resembles that data structure. Making the perceptron an object allows the framework to move the process around the network without technical difficulty, and also to do this without having to coordinate with the domain specific programmer through the use of extra interfaces. Each of the data flow objects has a module created by a basic user and a state object. The module contains an executable method which will be called for *n* iterations until the job is done. The state object stores the dependencies needed for computation. A design decision was made to separate the compute module from the data flow to allow a data flow to handle multiple modules. This allows the data flow implementation to support pattern operators[44].

To put it all together in one computation cycle, the data flow object will collect input packets from its connections. It will then call the compute method from the user's module, which will output a packet. The user's module may or may not store persistent data into the stateful object. At the end of the loop, the data flow object sends the output packets through its respective output connections. Figure 4.1 shows a data flow object, which contain:




```
package edu.uncc.grid.seeds.comm.dataflow;
public abstract class Dataflow  implements Data, Comparable<Dataflow>{
      protected HierarchicalSegmentID SegID;
      private HierarchicalDependencyID[] InputIDs = null;
      private HierarchicalDependencyID[] OutputIDs = null;
      protected Types.DataFlowRoll mDataflowRoll;
      private long CycleVersion;
      transient protected Dependency[] Inputs = null;
      transient protected Dependency[] Outputs = null;
      transient public long CycleVersionStop;
      transient private BasicLayerInterface UserModule;
      Serializable State;
      public abstract boolean computeOneCycle( );
      public abstract DataflowLoader getDataflowLoaderInstance();
      public abstract Dataflow getNewInstance();
      public abstract List<Dataflow> onGrainSizeSplit(int level);
      public abstract Dataflow onGrainSizeCoalesce
                    ( List<Dataflow> perceptrons ,  int level);
      public abstract SplitCoalesceHandler getSplitCoalesceHander();
}
```

FIGURE 4.1: The Dataflow class is used to represent the core entity needed in a data flow programming implementation.

- computeOneCycle() method is called to run the main computation method by the advanced programmer. In turn the advanced programmer should call the main computation method from the basic programmer's module.
- getNewInstance() method is used by the framework to create new data flows objects of the same data flow class.
- StateFul object holds the state data and the dependency information. The user should used this object to store any information that is needed between iteration of the main loop.
- HierarchicalDependencyID objects are used to hold the dependency information. Two arrays are used; one for inputs and one for outputs. More on Hierarchi-

calDependencyID is explained in Section 4.6.

- Dependency objects are used by the framework to create and maintain the connections once the data flow object is executed on the remote node. Notice that the objects are transient, which means that this data will not be sent over a network.
- HierarchicalSegmentID is used to identify the data flow perceptron.
- CycleVersion counts clock cycles needed to synchronize computation and communication over all the data flow network. Through the cycle version variable, the framework can coordinate computation halts while making sure no process falls out of sync and the original algorithm's behavior is preserved.
- OnGrainSizeSplit() method is called by the framework on a master data flow sink/source node to split the data flow perceptron into two or more child perceptrons.
- OnGrainSizeCoalesce() method is called by the framework to coalesce two perceptrons. The two perceptrons must be part of the same previously split perceptrons. That is, two perceptrons that do not share the same parent withing the Hierarchical tree cannot be coalesced.
- GetSplitCoalesceHandler() method is used by the framework to get a class that can be created by the advanced or basic user. The object, in turned, is used to split a Serializable data object into a list of Serializable objects. Since the basic user may be the only one with the knowledge of the data's contents, this user is usually the one expected to provide the split/coalesce algorithm.

An Unstructured template is used to load the data flows to each of the remote nodes.

In Seeds a parallel job is started by sending out a pattern advertisement. The advertisement has some information about the requirements a remote node must meet before it is allowed to help as an executor for the pattern.

The Seeds framework also has a MPI-like implementation whose boot-up process is similar to the data flow's process up to this point. The MPI-like implementation is covered in Chapter 2. Once the nodes receive the advertisement, they use Java Abstraction API to load up the classes that are specified in the advertisement. The classes are available at all remote nodes because they are included with the jar files that were sent to each server before the framework was started. Figure 4.2 shows a layered chart that summarizes the classes involved in this procedures and their order

1. Advertisement is Received: The Advertisement points Seeds to load up a BasicLayerInterface. This is the Object that was extended by the advanced user to cre-

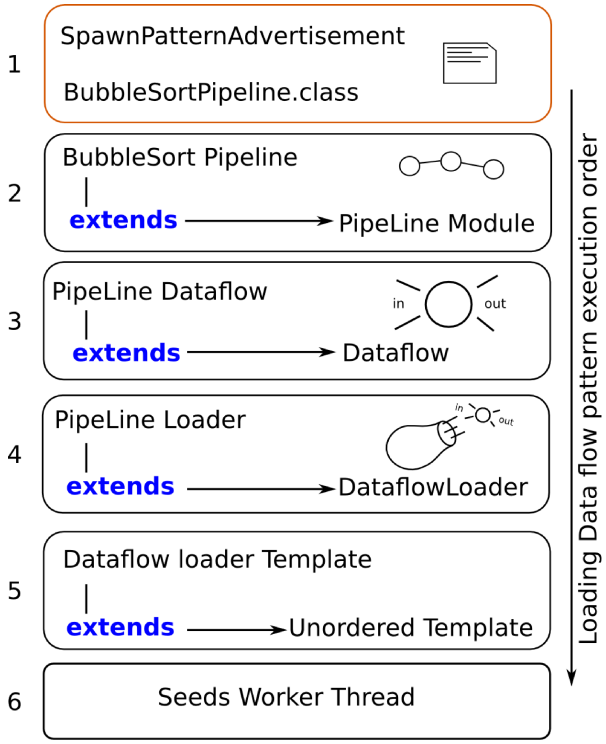


FIGURE 4.2: Classes involved in loading and operating a data flow pattern.

ate a parallel programming skeleton or pattern such as a pipe line in this example.

2. BasicLayerInterface Instantiated: The basic user implemented the PipeLine interface creating a module. The class name mentioned in the SpawnPatternAdvertisement is the basic programmers module.
3. Dataflow Instantiated: Once the BasicLayerInterface class (BubbleSortPipeLine in the figure) is loaded, it will have a static constant that points to loading a template, the template is a data flow implemented with the class PipeLineDataflow, which extends Dataflow class. The advanced user modified PipeLineDataflow class to manage the dependencies into a communication pattern that provides the pipe line behavior.
4. DataflowLoader Instantiated: The data flow in turn has a method that will return a new instance of the data flow's loader module. The loader module named DataflowLoader is extended by PipeLineLoader to load pipe line specific perceptrons. The Loader is responsible to create the different data flow perceptrons.
5. DataflowLoaderTemplate Instantiated: The loader module then is used by a DataflowLoaderTemplate. The DataflowLoaderTemplate is the main execution class for a data flow network. The template will connect to the master node in a pattern similar to a work pool skeleton. The master node will send the data flows to each of the remote nodes that subscribed to the pattern. Once the data flow pattern is loaded, the data flow loader will proceed to connect the input dependencies, and to publish the output dependencies. The last step for the data flow loader is to execute the data flow. At this point the data flow takes over and executes

the code that is pertinent to the pattern.

6. Execution Returns to Worker Process: When DataflowLoaderTemplate ends the main computation, it returns the perceptrons to the master node, and it returns execution to the process thread. This thread will wait in idle mode for a new pattern or for a shutdown advertisement.

The process of loading the data flow pattern allows each of the steps to be administrated by a class, and the classes in each step can be extended to modify a pattern at the advanced level, and at the basic level. Reflection code is not part of the high performance parts of execution to prevent it from affecting performance.

4.2 Advanced User Layer Creating a Pattern Using Data Flow Seeds

At the advanced user level, the programmer starts up with implementing a data flow template. The implementation of a pattern from this perspective consist of developing a pattern loader, and a data flow. The pattern loader has two signature methods:

- Dataflow onLoadPerceptron(int segment): This method is called to get a data flow perceptron from the advanced user. Using the implementation of a pipe line skeleton as an example, the perceptrons loaded for this skeleton would consist of the stages that this data flow would have to compute. The segment integer tells the advanced programmer which of the data flows the framework is loading at the moment. The integer goes from zero to *n* where *n* is the least number of compute nodes necessary to start the pattern not counting the source and sink tasks. The source and sink perceptrons are requested using arbitrary negative segment numbers (-12, and -13 respectively).
- void onUnloadPerceptron(int segment, Dataflow perceptron) : the method is

called to return the data flow to the advanced user. In the example of a pipeline, this method is not used, but in the case where a stencil pattern is implemented, the method is used by the advanced user to return the stateful data to the basic programmer. This varies because some algorithms value the data streamed from the source to the sink, and some other algorithms value the data that is computed on the stateful objects after all the streamed data has been used to compute onto it. Section 4.3 helps explain this point further.

4.3 Stream Input data and Initial State Data

The framework deals with two inputs for its data. The stream input data refers to a constant feed of data that will be going on throughout the lifetime of that parallel computation. In the pipeline example, this refers to the packets of data that are fed in to stage one, and that are then streamed through each of the stages. The initial state data refers to data that is necessary as an initial state for the program to execute, but which does not need to be updated from an external source throughout the computation. A good example of this is the initial matrix state for a 2D stencil, or the stage numbers allocated to each process in a pipeline. Seeds provides the advanced user with the initial state data through the use of the signature methods previously mentioned. The stream input data is optional to the advanced user, since some of the patterns don't make use of it. If the advanced programmer needs the streamed input feature for a skeleton or pattern, it can be enabled by returning true in the signature method instantiateSourceSink.

Figure 4.3 shows a diagram on how the classes implemented by the advanced user interact to run the data flow pattern. The class DataflowLoaderTemplate retains execution throughout the life of the data flow pattern. DataflowLoaderTemplate uses the DataflowLoader class to communicate with the advanced user, and obtain the data flow

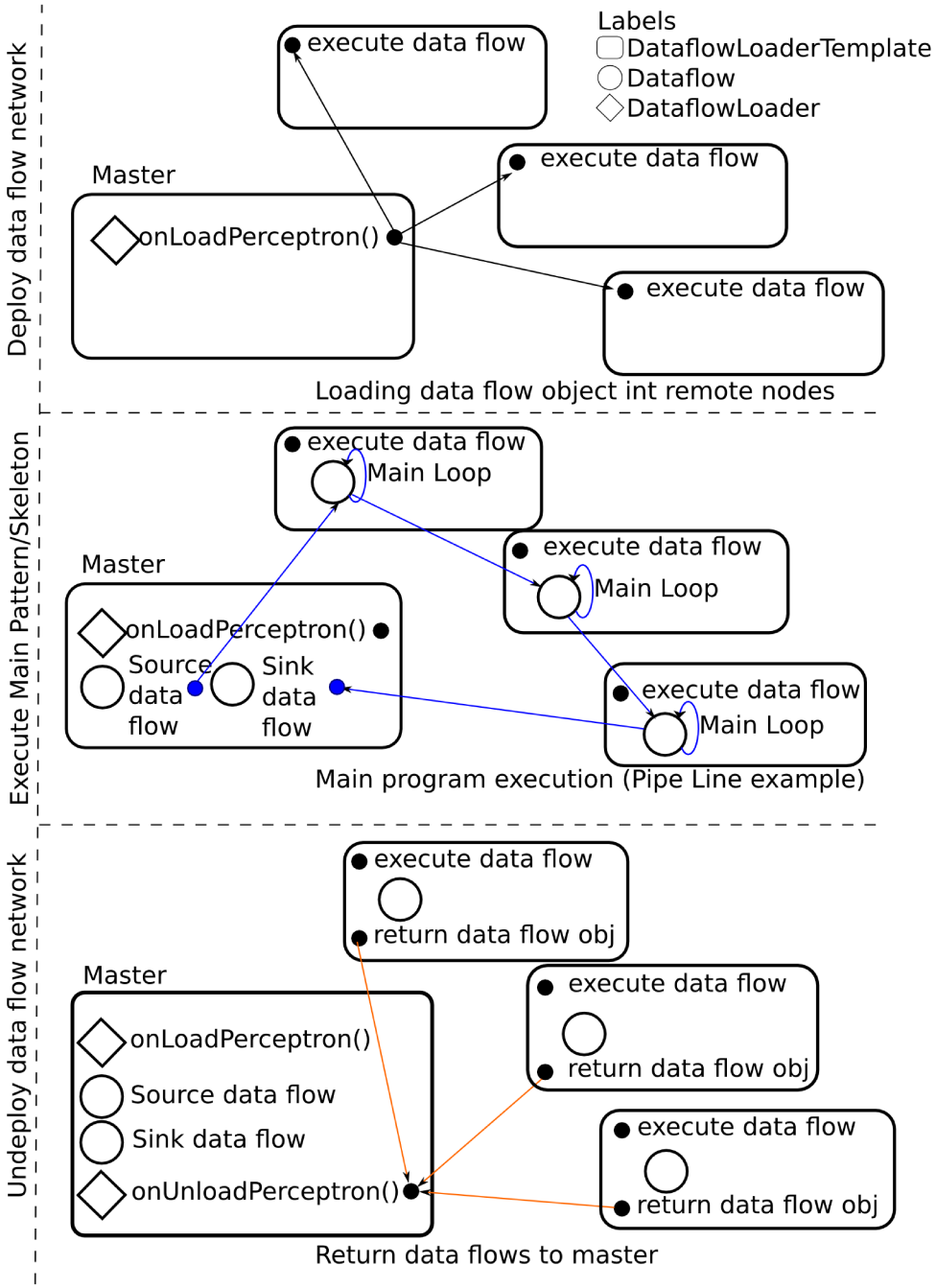


FIGURE 4.3: The steps involved in deploying, executing and un-demploying a data flow network.

perceptrons that are then sent to each of the remote nodes. The second step is to execute the data flows. The main loop is also controlled by the DataflowLoaderTemplate. In it, the Dataflow class is called to execute. It takes in input packets and outputs output packets after it has done one iteration. This is repeated for some number of iterations. The last step is to return the data flow objects to the master node using onUnloadPerceptron(). The DataflowLoaderTemplate will call this method from the DataflowLoader class. Finally, the DataflowLoaderTemplate returns execution to the main worker thread, which will wait for a new pattern advertisement to start the next job or a termination advertisement.

4.4 Hierarchical Dependencies

Up to this point, the Seeds framework is implementing work previously researched by other projects[59], [57]. This section presents the extension to the dependency concept that enables the data flow based framework to provide automatic scalability and grain size. The main extension to the dependency concept concerns with splitting a dependency. The extension does so while providing the benefits of automatic scalability to the advanced user through an interface that is easy to understand. To some extent, the feature can be provided to the basic user either with no extra effort, or through implementation of a couple of extra signature methods.

4.5 Dependency

A dependency is a stream of data that will flow from one perceptron to the other. The dependency describes a stream. Once connected, the dependency can be used for the rest of the computational time. The dependency must be identified uniquely, and its identification should be independent of the hardware where the process is running. This is done

to allow the framework to move the process and the dependency for auto-scalability and grain-size, although the feature also helps to provide fault-tolerance and it is required for load balancing.

The protocol that is used to request a dependency is shown in Figure 4.4. The dependency client will scout the peer-to-peer network for an dependency advertisement that has the specified ID. Once found, the client will connect to the dependency server. In the handshake packet, the client includes the ID to which it wants to connect. The dependency emitter checks the ID against a map of hosted IDs. If present, the dependency server will return a positive acknowledgment; otherwise, the dependency server returns a negative acknowledgement. Each of the perceptrons has a dependency engine, the engine is used to "publish" the output that the perceptron will be producing. Once connected, any other request for that dependency ID is denied.

The dependency engine mainly abstracts the process of finding, and handshaking a connection. The dependency engine in turn relies on the MultiModePipe abstraction implemented for MPI-like Seeds.

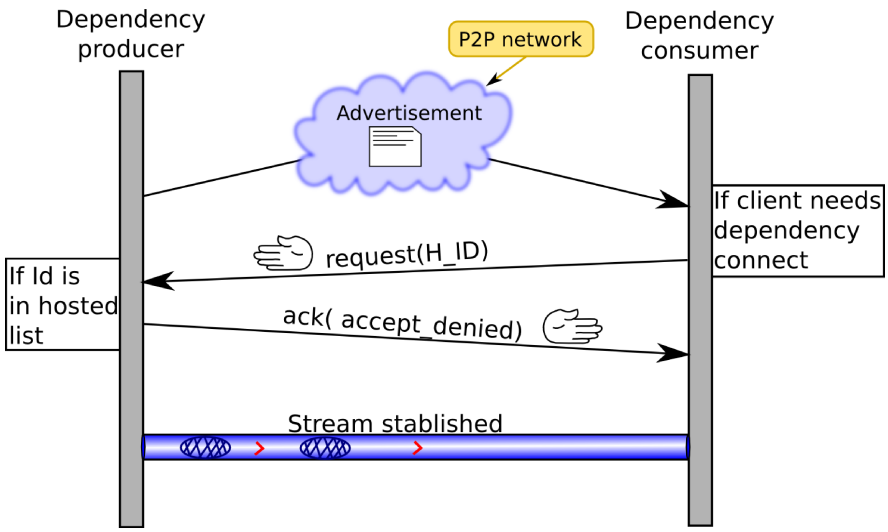


FIGURE 4.4: The protocol used to connect to a dependency.

4.6 The Hierarchical Dependency Syntax

Each of the dependencies must be assigned a unique ID. However, simply using an integer is not feasible because more dependencies can be created by individual processes, and the hereditary history of a dependency must be maintain in order to allocated a dependency to its corresponding output process. For example, suppose a dependency *A* is split into *B* and *C*. Now, there should be a process by which a perceptron connecting to dependency *A* would know that by connecting to dependencies *B* and *C*, it can produce dependency *A*. The process should also indicate how many child dependencies are needed to put the parent dependency back together.

In order to accomplish this, the dependency ID is an integer followed by a slash (/) followed by the number of dependencies that make up the whole. For example, if we have a dependency 2 that is part of a family of 4 dependencies, its ID would be expressed as :

- 2/4

In the implementation, the root dependency is always part of a family of one dependency (n/1). Its children then are attached to the root dependency with a dot. For example, we can create a dependency 2/4 that is child of 1/1. The id is then expressed as

- 1/1.2/4

This means that, in order to receive dependency 1/1, a perceptron must connect to:

- 1/1.0/4
- 1/1.1/4
- 1/1.2/4, and
- 1/1.3/4

Figure 4.5 shows the basic single level configuration for a binary dependency. In this simple configuration, one can observer three possible arrangements. On the left, we can use the dependencies to establish a point-to-point connection with the dependency source symbolized by a wave, and the dependency consumer or sink symbolized by a ground symbol. The second arrangement, in the middle, can have a dependency splitting itself so that the original output can be split and sent to two remote nodes. The split action is done using an interface implemented by the advanced or basic programmer. The last arrangement, on the right, is a dependency consumer that requires dependency 1/1, but can only find the partial dependencies in the network (1/1.1/2 and 1/1.2/2). The consumer can connect to both of these dependencies and coalesce the input to create the needed data.

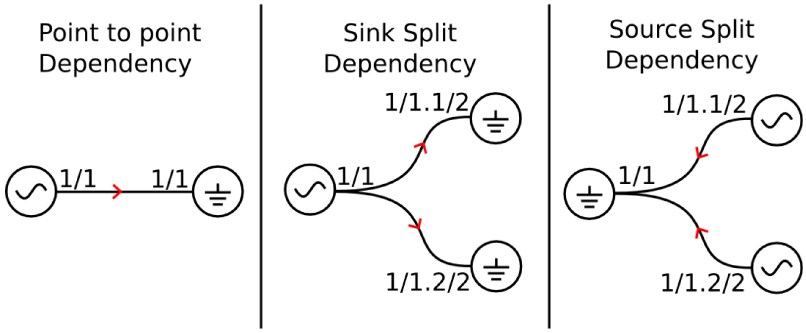


FIGURE 4.5: Basic point-to-point, sink-split and source-split dependencies.

The syntax allows the dependencies to split multiple times. For a more concrete example, suppose we have an n-body problem of 20 particles distributed among five processors. Figure 4.6 shows the initial data flow network created to solve this problem on the left. After some iterations, suppose the framework splits one of the perceptrons into two child perceptrons to take advantage of a new sixth core. Figure 4.6 shows how the dependency is split on the right side. From the perspective of the basic programmer, this means now, particles 1 through 2 will be updated using dependencies A through E, and 3

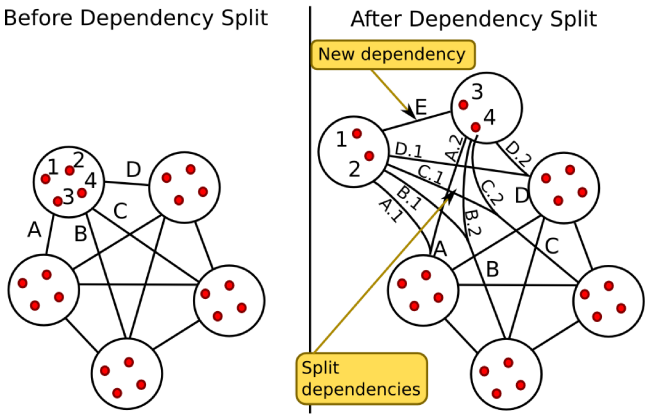


FIGURE 4.6: Example of hierarchical dependencies splitting in an all-to-all synchronous pattern.

through 4 will be updated using the same dependencies. Depending on the pattern, the advanced user may be able to resolve a split without additional input from the basic user. However, in the case of an all-to-all pattern and 2D stencils, some of the job would need to be transmitted to the basic layer programmer using interfaces and signature methods. For the all-to-all pattern, C.1 and C.2 have the same data as C because both of the new processes require all of the data to continue computing.

Another feature provided by this extension is that only the split perceptron need to be of concern. The other processes would see little change at the advanced level and basic level. For example, the perceptrons that were not split will still need the dependencies from the split perceptron; however, the hierarchical dependency will add the output from C.1 and C.2 to make the output for dependency C. This example assumes the dependencies are duplex to keep the example simple. The actual implementation would use two dependencies, one going each way.

4.7 The Hierarchical SegmentID

As it can be seen in Figure 4.6, the local data, or stateful data is also split. The number of particles each of the child process gets is split into two each. To keep track of this, the

perceptrons are also assigned a ID that complies with the same syntactical rules explained for the hierarchical dependency. A complete dependency ID is therefore made up of the hierarchical segment ID plus the dependency ID. The two are joined by a colon (:). For example:

- 2/1:0/1

The hierarchical segments are the root ID in the process arrangement in the minimum CPU configuration. The number of CPU's in the minimum CPU configuration is specified by the advanced or basic programmer.

4.8 Technical Details

This section will explain some academically important algorithms and protocols. In order to provide enough information to explain them, some technical information about the framework is presented. The hierarchical dependency concept is implemented using multiple Java classes.

- The HierarchicalDependencyID implements the syntactical rules used to describe the hierarchical dependencies. It uses recursive methods to allow for the creation of infinitely nested hierarchical ID's. The class is used in the data flow perceptron to carry with them the basic information needed to establish a dependency stream.
- The HierarchicalSegmentID represents a hierarchical segment ID. It is included as part of a hierarchical dependency. It should be noted that the term segment is used by the framework to denote a piece of stateful data. One must avoid confusion by understanding that a stateful data can be identified by a segment, which is an integer ID, and it can be identified by a hierarchical segment, which is the nomenclature explained in this section, and also uniquely identifies a stateful data

piece.

- The DependencyEngine class is used to separate the communication specific side of the implementation from the process of finding and establishing the connection. The Dependency Engine class has a reachable socket server or shared memory data structure that is used to accept connections to dependencies hosted at the local process. It also has a method to find and connect to an output dependency.
- The Dependency class, at the basic level, is a wrapper for a stream object. It allows the programmer to send data to a client process. Note that a dependency is one way, so in order for two processes to have a duplex connection under this implementation, each must publish an output dependency, and connect to the other's dependency. This class also has recursive methods to send data on a split connection, as well as an algorithm to aggregate information from multiple streams to be fed to as a single dependency for a local process.
- Hida: The name abbreviates Hierarchical ID Dependency Advertisement. This class is used to organize Dependency advertisements so they can be searched efficiently at the client side of the connection. The object manages a recursive tree structure. On the server side, a hash map keeps track of the hosted dependencies.
- The DependencyMapper is a static class that will manage the Dependency advertisement using the Hida object. One DependencyMapper instance is used to service all the processes on the node.

The Dependency implementation has a tree-like structure which is traversed using recursive methods. A Dependency therefore has an array of Dependencies called Children.

Most of the methods implemented in the Dependency class can recurse over the dependency tree to return if the dependency has data left, if the dependency is connected, etc. The two main methods where most of the conceptual and technical details concentrate are takeRecvObject() and sendObj().

4.9 The Receive Method

The receive method returns the next packet that has been received from the remote process. The receiving algorithm has to integrate streams from multiple sources. Figure 4.7 shows a decision diagram on how a packet is handled by the hierarchical dependency on the left. On the right, the figure shows a packet traveling up the dependency tree before being handed of to the local process. The tree on the right shows all the possible routes the decision tree may take. The dash lines connect the decision from the decision tree that correspond to the path traveled by the packet in the diagram on the right.

Following the decision paths from Figure 4.7, if the stream is null, we assume that it is

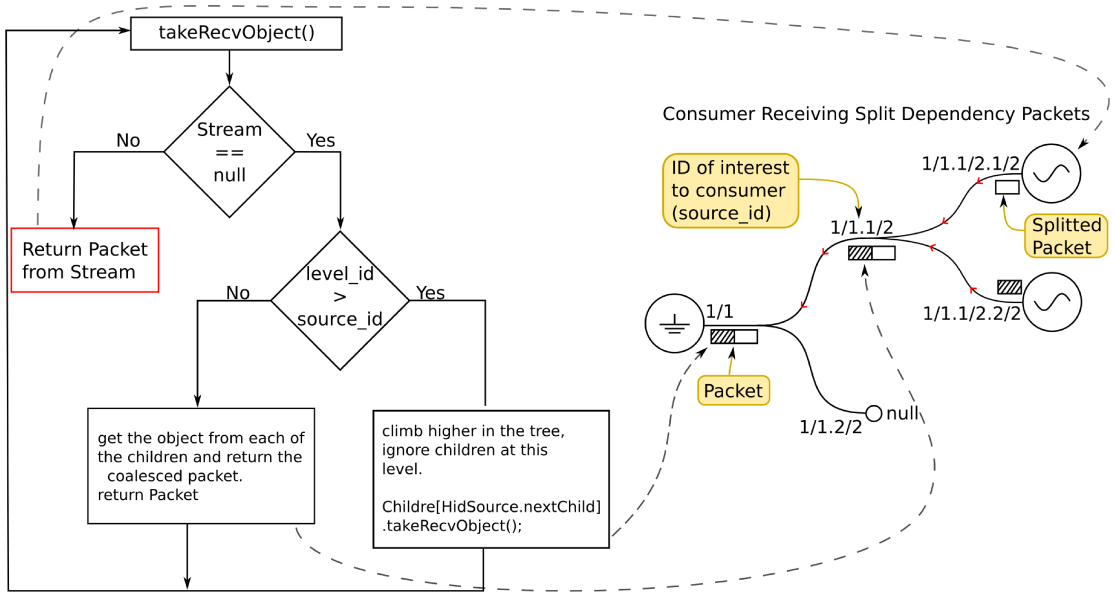


FIGURE 4.7: A decision tree to handle a packet on the hierarchical dependency receive method (left), and diagram showing how a packet travels down the dependency tree for a dependency with binary split sub-dependencies (right).

because the connection is made up of multiple streams, and each of the children contains a stream. Then, we expect each of the children will return an object, each with the same version, which can then be submitted to an advanced or basic user. The stream is submitted to the advanced or basic user through a method that returns the coalesced packet for this dependency.

Each Dependency object has a level_id variable. The level_id is a HierarchicalDependencyID that is used to label the level in the tree. Likewise, the source_id is also a HierarchicalDependencyID. The source_id is used to store the dependency ID that the dependency tree should be servicing to the client connection. A comparison is made between the level ID and the source_id objects. To say *x* is higher than *y* is to say that *x* is a child of *y*. if *x* is lower than *y*, is to say *x* is a parent of *y*. In the decision tree, the comparison is used to determine if the children connections should be used to get the next object. Since each of the Dependencies starts at a root, in many cases, the dependency that is requested does not fall all the way down to the root, and therefore many of the children connections are not needed. The corresponding example from the figure is dependency 1/1:2/2. Because the dependency being serviced is 1/1:1/2, dependency 1/1:2/2 is not necessary, and waiting for it can deadlock the connection. If the level is higher than source_id, the algorithm keeps climbing up the dependency tree ignoring the other child dependencies. If the level is lower than the source_id, the algorithm retrives the object from the stream of each child, and coalesces the data objects into one object. Finally, the algorithm returns the object.

4.10 The Send Method

The send method sends a packet to the remote node. Figure 4.8 shows a couple of dia-

grams. On the left is a decision tree used to handle packets send through the dependency. On the right is a diagram that shows the path taken by a packet on its way to being sent through a binary split dependency. Like the takeRecvObject(), this method assumes the Dependency is a root if the Children array is null; Otherwise, we look at the level_id of each dependency on the way up the tree to judge when we should start splitting the packet. At some level, the packets starts to be split; however, the remaining part of the packet can keep climbing the tree until it reaches a leaf Dependency that will send the packet over the network.

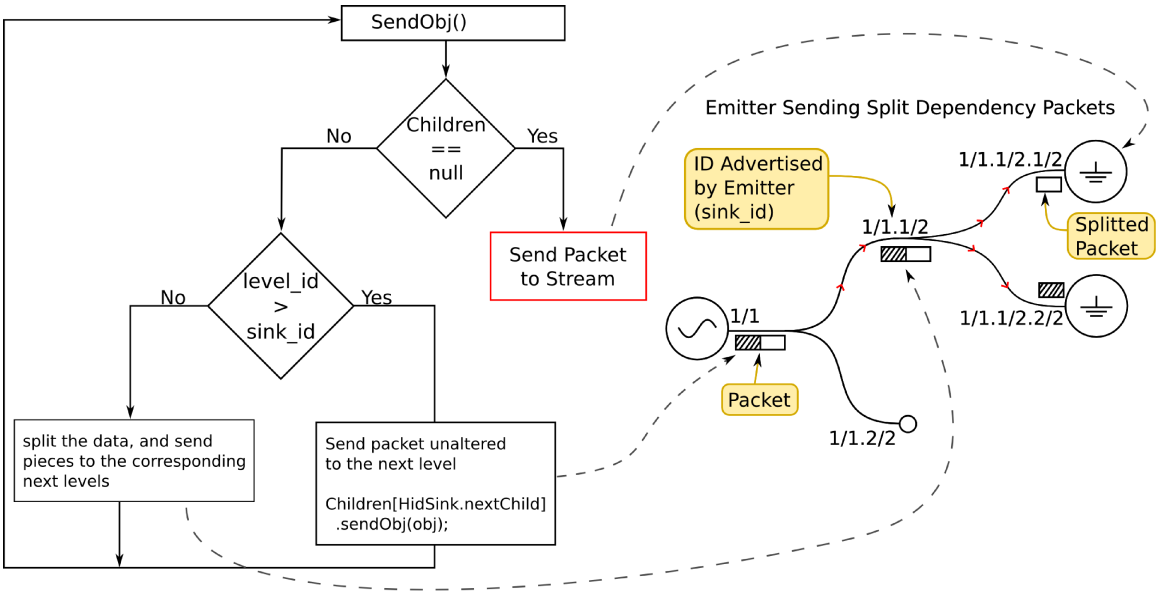


FIGURE 4.8: A decision tree to handle a packet on the hierarchical dependency send method (left), and diagram showing how a packet travels up the dependency tree for a dependency with binary split sub-dependencies (right).

4.11 Hibernating Dependencies

When a data flow is selected to return to the master node to be split, we call that state a **hibernation state**. To accomplish this transition, the framework must be able to freeze the computation node without having knowledge of what the computation node is doing inside. We added three features to enable the framework to hibernate the dependencies.

We use a version number on the messages that are transferred among the data flows, and we add an algorithm that will negotiate a shutdown at some version in the future. First, we require each of the messages between the computation data flow to be stamped with a version number. Using a version number allows each data flow to verify that the messages are the next expected. This also allows the data flows to synchronize and adds about 32 bits of bulk to Seeds' communication layer. Alternatively, the implementation can only use the version number on each of the data flows without sending an asserting integer to the remote nodes. This is studied further in Section 5.2.

The second feature is a protocol to negotiate the hibernation state at some version in the future. Figure 4.9 shows the protocol in a basic form. The protocol will compute a version in the future at which point the local data flow and the connection of its neighbors that are involved with the local data flow would disconnect. Notice that the framework could arrange to hibernate a perceptron at an iteration in a future where the parallel application could have finished computing. This is acceptable because, if the computation is finished, there is no need for the rearrangement. However, if the program is still running

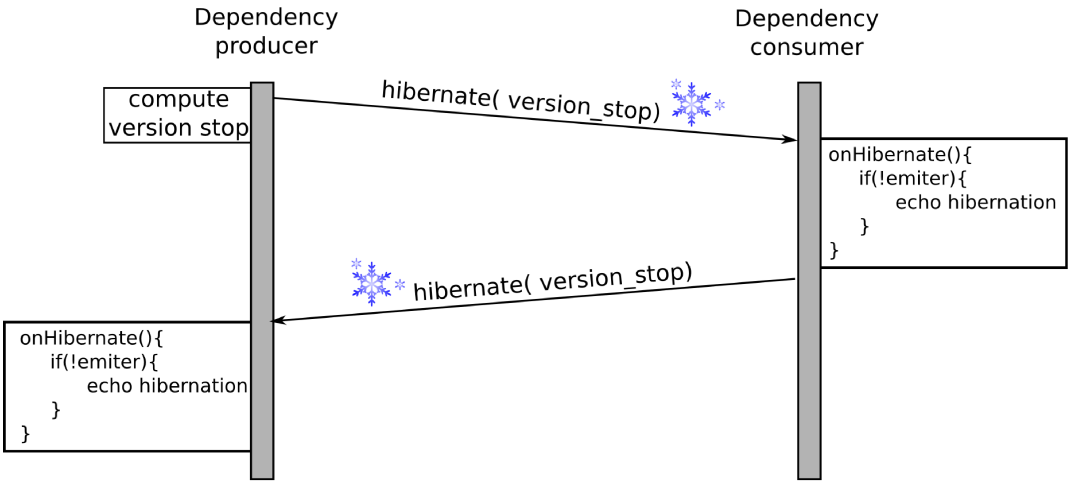


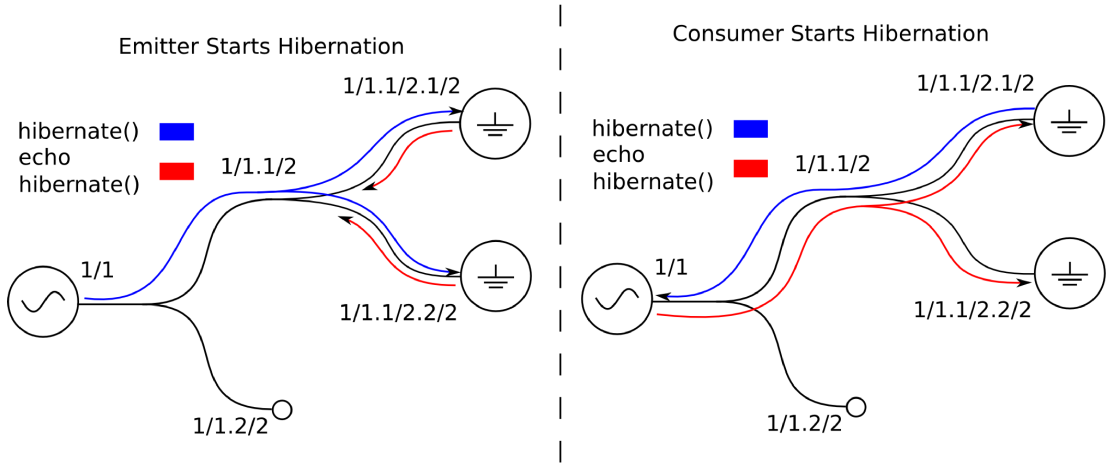
FIGURE 4.9: Point-to-point dependency debating a future hibernation point.

at that future iteration, the program can benefit from the increase use of resources.

A point where the perceptron will hibernate too close to the end of the computation to bring any speedup could happen under this implementation. Although hibernating close to the end of the parallel application can slow down the speedup, it does not destabilized the application either, and a modification to this algorithm can prevent the application from hibernating if the iteration is too close to the computation's final iteration. This could be studied in future work.

The protocol then insures that a hierarchical dependency is hibernated by echoing the message. Figure 4.10 shows how the protocol behaves on the dependency tree. The message must be echoed in order to spread a hibernation call throughout the dependency tree. If a source emits the call, the sinks will adopt the version stop set by the source, and will echo the call with the version stop number. The echo stops once it comes back to the original emitter. The example where a consumer starts the hibernation (on the right) better shows the need for an echo. In this example, the neighbor sink will only know of the impending hibernation when the source echoes the hibernation call.

FIGURE 4.10: A hibernation call traversing a hierarchical dependency tree.



The echo allows the hibernation message to go to all the leafs of a dependency tree. Also, the emitter of the hibernation call will not start the procedure itself. It waits for the message to make a round-trip. However, given the strict timing demanded by the advance negotiation algorithm, care must be taken to ensure the processes do not fall out of sync and deadlock.

To prevent deadlock, the algorithm in Figure 4.11 shows how the version stop is negotiated between the local process and the neighbor remote processes. There are a few variable used to negotiate and monitor the hibernation process:

- *current_version:* The version being executed at present by the local process.
- *version_stop:* The version number at which hibernation will happen. The hibernation is only done by the perceptron with the hibernation action. And the perceptron that shares a neighbor connection with the hibernation perceptron also gets notified of the hibernation. Although the neighbor perceptrons don't hibernate, their connection do disconnect when version_stop is reached. Posterior to that, the neighbor perceptron reconnect to the dependency ID.
- *wain_n_version:* The version number at which point the perceptron can reattempt a hibernation action. This can happen if other perceptrons are already scheduled to hibernate.
- *lowerend*: A version that should be less than *version_stop*. The variable is used to asses if the local process can join a simultaneous hibernation action along with a neighbor perceptron. If the version_stop of the local process is lower than lower end, the process must wait further to attempt the hibernation call again.

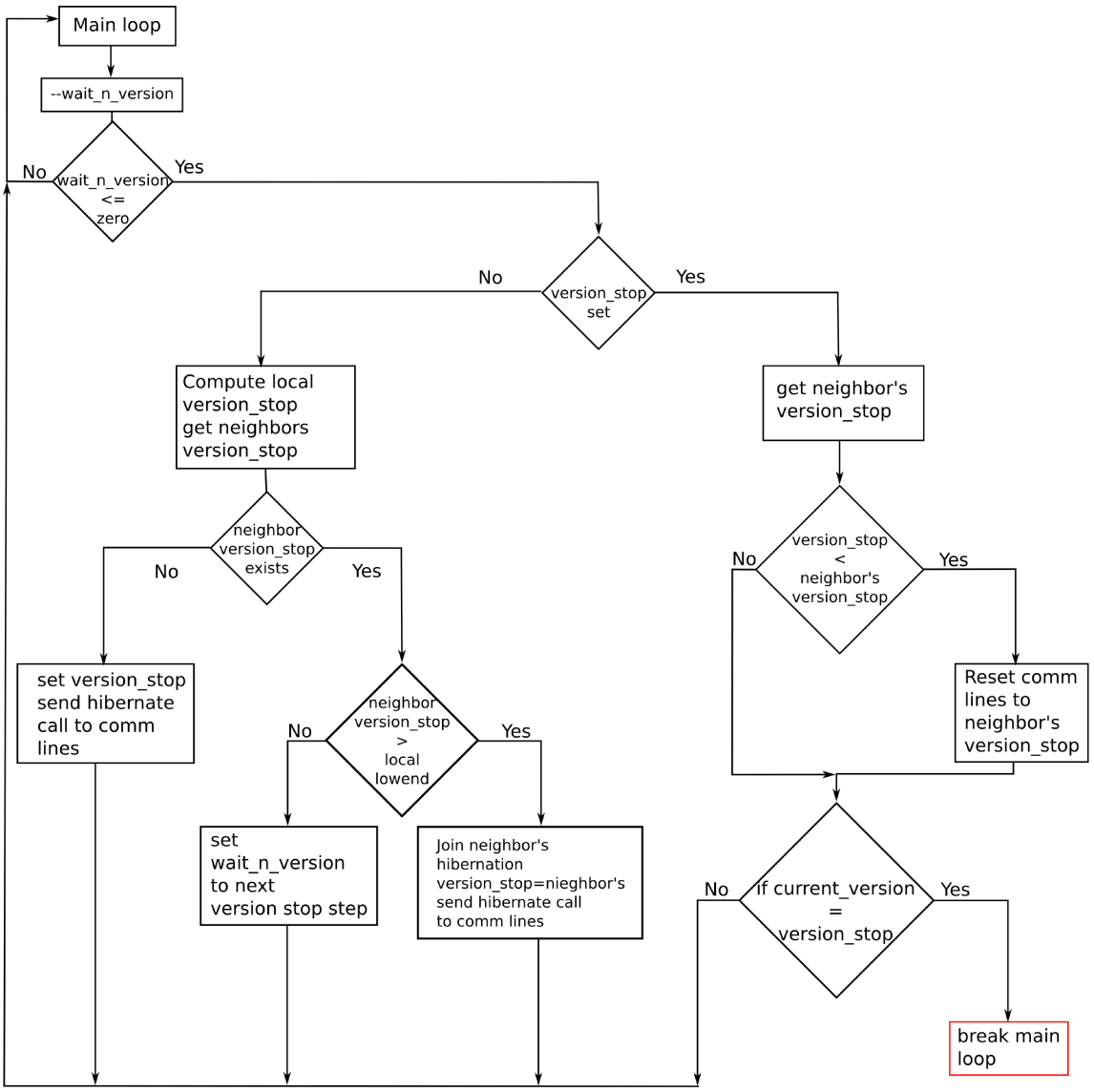


FIGURE 4.11: Decision tree to negotiate a hibernation state for the local process.

First the main loop enters this negotiation once the advanced template or the framework sets the data flow status as hibernation eminent. This starts the negotiation process. The algorithm checks to see if *wait_n_version* is set. If so, this means that a neighbor process is close to entering hibernation. Therefore, it would be unsafe for the local process to continue with the hibernation procedure. If *wait_n_version* is not set, the *version_stop* is checked to validate if it was set in a previous iteration. If *version_stop* has not been set, the procedure will calculate a *version_stop* based on the speed at which the

local process has been working through the previous iterations. The number is then compared with the *version_stop* of the neighbor processes, which should not be set most of the time. If the neighbors is not set, the local process will send the hibernate call through its communication lines. If the neighbor's *version_stop is* set, the local process has to judge if the *version_stop* of its neighbors is far enough away into the future to join the group and hibernate at that same time. *lowerend* is calculated for that purpose. If the neighbor's *version_stop* is larger than the local process' *lowerend*, the local process will join the neighbor's hibernation version. If not, that means there is not enough time (measured in cycle units) to organize a hibernation for the local process. The local process will then set *wait_n_version* to a save version in the future and continue computing until that time.

Once the *version_stop* variable is set, the process will continue to enter the decision tree. This time it will go through the branch where *version_stop* set is true. At this point, the local process will still check on the neighbor's *version_stop*. Although unlikely, the check is useful in case the neighbor's and the local process initiate a hibernation call at the same time, but were not aware of each other due to delays in the network. If this happens, the processes should reset the hibernation call to the highest version among the group of hibernating processes. Notice that this will not lead to an infinite loop where all processes continuously update the hibernation version further and further into the future. This will not happen because any new process that enters the hibernate procedure will, at some point, be aware of its neighbor's hibernation state before issuing the hibernation call, and therefore will instead enter into the the tree branch that sets *wait_n_version* vari-

able.

The last point in the decision tree is once the *current_version* reaches the *stop_version*. At that point, the main loop is broken, and the data flow can be disconnected and returned to the master node.

The *version_stop* is calculated by measuring how fast the local process is going through the iterations for the main computation loop. The aim of the algorithm is to set the hibernation action far ahead in the future so that there is enough time for the network to synchronize around the action.

The variables mention in the algorithm are computed using multiple parameters gather from by the framework. The variables used are:

- t = time taken to compute i iterations.
- i = number of iterations done.
- *ss* = time slot size in millisecond units. The time slot is a constant based on the highest latency expected from the network.
- *vts* = version per time slot. The number of versions done in one time slot.
- *cqs* = communication queue size. The undelivered or unsent packets is assumed to be the highest by using the maximum number allowed to be cached in the send and receive queues.
- s = time slots.
- STEP_SIZE = defines the ceiling of the next multiple of STEP_SIZE. This means only multiples of STEP_SIZE are allowed to improve the change that unsynchronized nodes will hibernate on the same iteration.

Now we can calculate the variables without a given value:

$$s = \frac{t}{ss}$$
$$vts = \frac{i}{s} + cqs \tag{4.1}$$

Now, *vts* can be used to calculate *version_stop* and *lowerend*.

$$\begin{aligned} version\_stop &= current\_version + vts * a \\ lowend &= current\_version + vts * b \\ wait\_n\_version &= current\_version + STEP\_SIZE + vts * c \end{aligned} \tag{4.2}$$

The constants a and b are parameter chosen to provide a tolerance. *a* provides tolerance between the present version and the hibernation version, and *b* provides tolerance between the stop version computed by the local processes and the version computed by the neighbor process. If the tolerance is too low, the algorithm will fall out of sync, if the tolerance is too high, the parallel application cannot react quickly to a changing environment. *c* provides tolerance for the future attempt at hibernating, since the first attempt failed.

The environment does play a role. For example, the tolerance for a cluster computing environment should be larger than a shared memory environment because the delay increases. A shared memory system can have a smaller tolerance. In a mixed environment, the tolerance should be set considering the highest delay expected.

The end result for the hibernation algorithm is that the perceptron can be taken off-line safely. The perceptron can then be split into more processing units or it can be coalesced with other perceptrons to have less computation processes.

The remote processes that only had their communication lines hibernated but where not hibernated themselves will try to reconnect after the hibernation call has been executed. Once the new perceptrons are on-line, they will emit dependency advertisements that

will be caught by neighbor perceptrons in the network that were already looking for the connection. Once the connections are reestablished, the perceptron can resume computing.

4.12 Results

The hierarchical dependency concept was tested on the Seeds framework using three algorithms. The first two algorithms were a bubble sort algorithm and a matrix multiplication algorithm using the pipeline skeleton. Those implementations were done with the aim to measure the performance overhead incurred by the hibernation action and by the split algorithm on a binary, one-level hierarchical link. The third test involved measuring programmability. The heat distribution algorithm implemented with a stencil pattern was used. A data flow stencil was implemented for this purpose, and signature methods were added to the stencil pattern in order to enable the automatic scalability feature.

For all three tests, the hibernation action was triggered statically by a hard-coded split action inserted into the pattern template. In a production environment, a load balancer would make the decision to split a perceptron based on performance measurements done to the application. The load balancer would then send a control message to the selected perceptrons notifying them of the action.

4.12.1 Bubble Sort using a Hierarchical Dependency Pipeline

These preliminary results measure the hierarchical dependency's ability to adjust to a changing number of cores. This first test does not test the use of the dependency split feature. It only measures the overhead incurred when a perceptron is taken off-line and split into more perceptrons. The creation of new dependencies is done so that the computational task can be divided into two parts with the new dependency connecting the two.

A parallel implementation for the bubble sort algorithm on a pipeline skeleton was used as the main problem[51]. The algorithm can be implemented having one pipeline stage for each of the indexes in the lists. If a list of lists is in need of sorting, each stage can be working on bubbling up one number from the list per process. In a list of 5000 numbers, there can be 5000 stages and therefore as many processes working on it. Multiple stages can be assigned to one process in order to reduce the number of processes needed. The implementation for Seeds can perform this task automatically without changes to the basic programmer's code and it can do it during runtime.

Five thousand lists of 5000 number were sorted using a skeleton interface identical to MPI-like Seeds and similar to skeletons done by previous work [57]. The time taken to go through the list was measured to judge the impact incurred by the overhead of organizing processes to be added to the computation, or taken out.

Figure 4.12 shows the results for Coit-grid Shared Memory. Static means that the program ran from start to finish with the same number of processes mentioned in the *x* axis. There were three dynamic runs. The first dynamic run scaled from two cores to sixteen cores. That means that out of the 5000 iterations, the first thousand iterations where done using two processes, the second thousand were done using four, the third thousand iterations where done using 8 processes, and the final two thousand iterations where done using sixteen processes. The four core dynamic test scaled in the same way but starting from four cores; that is, it scaled from four to eight cores and from eight to sixteen cores to complete the test. The eight core dynamic test scaled from eight cores to sixteen cores. The ideal bar is a number calculated using the information from the static tests, and math-

ematically calculating the time it should have take for the dynamic tests. Equation 4.3 shows the calculation for the test starting with two cores and finishing the computation with sixteen cores. *S* represents the static run. The subscript represent the number of cores used, and *I* represents the ideal performance for the test.

$$\frac{1}{5}S_2+\frac{1}{5}S_4+\frac{1}{5}S_8+2/5\,S_{16}=I \tag{4.3}$$

I gives an approximation of how long it would take the program to finish if there was no overhead while still having the increase in processes every thousand iterations.

As Figure 4.12 shows, the Coit-grid Shared Memory was able to take advantage of the increase in resources and improve the program's running time for two and four threads. At eight threads, the overhead of reorganizing the processes adds more time than it would have taken had the program run on just eight processes from beginning to end. Improvements in implementation can reduce the overhead. Overall, the feature adds an average

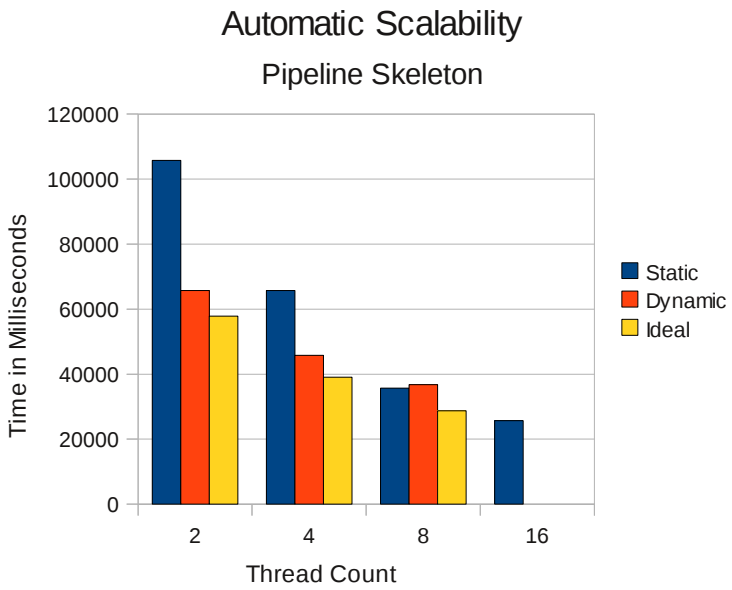


FIGURE 4.12: Bubble sort using a pipeline on Coit-grid Shared Memory.

14% overhead when compared with the optimal number, which provides more information about performance than the static measurement. The static measurement is provided to give an idea of how long it would take the application to finish if it stays with an static, no-extra-overhead run. The lowest overhead was 5% for the two-core tests, and the highest was 28% for the last test. It should be noted that the two-core test incurred the split overhead three times where the eight-core test only incurred the split overhead once. The actual overhead measurement for the two-core test is 14%, but that accounts for all three split sessions. In contrast, the eight-core test only has one split, and its actual overhead was 28%.

It is left as future work to find out if grain-size affected the eight-core test. At sixteen cores, each process had been working over 312 indexes. Another characteristic that adversely affects the results is the use of hyper-threading threads. Ideally, the experiment would have used only cores since multiple threads on hyper-threaded cores are architecturally different than a thread per core.

The same experiment was performed on the Coit-grid Cluster. This time, the network delay had an effect on performance. Figure 4.13 shows the results for the distributed memory system. The results show that the overhead is too costly for this program in particular. For two, and four processes, the static solution would have finished faster than the dynamic solution that had access to more resources over time. The average overhead for this experiment was 16% with the highest being 18% for the four and eight-core tests, and the lowest being 12% for the two-core test. The eight-core test on the cluster, in contrast with the same experiment on the shared-memory system was the one that performed

best.

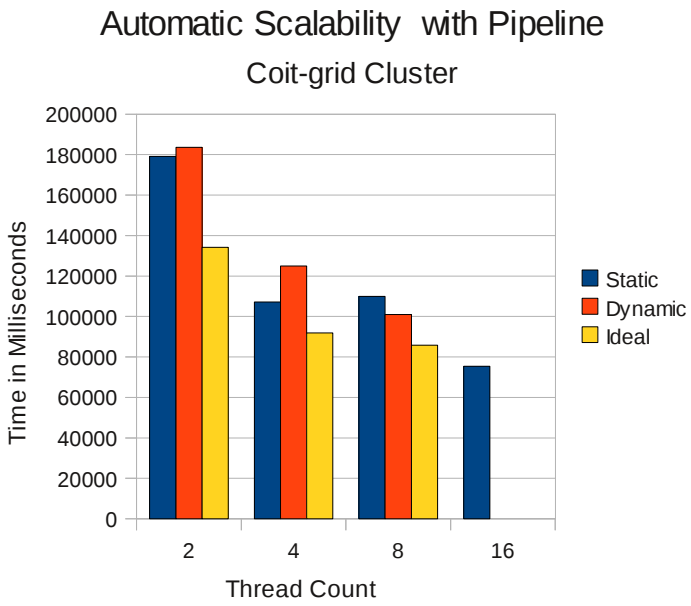


FIGURE 4.13: Bubble sort using a pipeline on Coit-grid Cluster.

4.12.2 Systolic Matrix Multiplication using a Hierarchical Pipeline

This section presents a short review of matrix multiplication, a short review on implementing matrix multiplication using a systolic array. The results show that adding automatic scalability to the sideways split pipeline implementation adds 82.93% more non-functional code to the implementation, and its scalability is competitive with the scalability of simply scaling the pipeline by distributing the stages among more processes as was done in the test using the bubble sort algorithm.

Matrix Multiplication

The matrix multiplication algorithm was chosen because it showcases the use of automatic scalability, and because the algorithm can be implemented using a pipeline. The pipeline is helpful because the data flows one way, which allows the implementation of

automatic scalability to be coded in steps that can be tested. The matrix multiplication was implemented as mention by Wilkinson et al. [51].

In matrix multiplication we have A x B = C. The basic algorithm for matrix multiplication is shown in Figure 4.14.

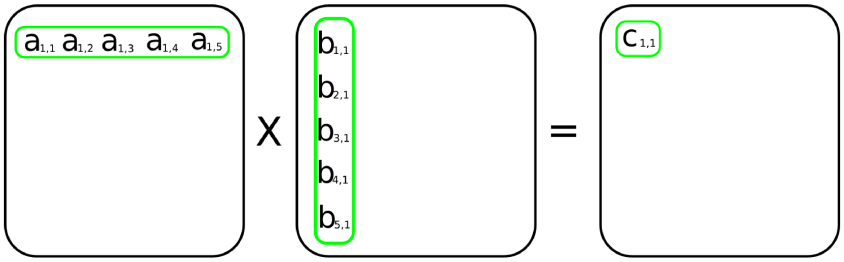


FIGURE 4.14: Matrix multiplication algorithm

Equation 4.4 shows the computation done to row one of matrix A and column one of matrix B to get the value for $C_{1,1}$

$$a_{1,1}b_{1,1}+a_{1,2}b_{2,1}+a_{1,3}b_{3,1}+a_{1,4}b_{4,1}+a_{1,5}b_{5,1}=c_{1,1} \quad (4.4)$$

In the pipeline implementation, A is divided into its columns, and each column is assigned to a stage. We differentiate stages from processes here because there can be more than one stage per process in our implementation. The matrix B is divided by its rows. The rows from B are then piped through each of the pipeline processes, and each process pipes the row through each of the stages. At the end of the computation, the returned columns correspond to the columns of matrix C.

Automatic Scalability Experiment's Implementations

The matrix multiplication algorithm was implemented in three forms, with the last two requiring infrastructure modification on the framework to get the prototypes ready.

Serial: The serial split is a pipeline with only one process working on all the stages.

The source and sink for matrix B's rows are hosted on a different core or host computer. This version is held as control. Figure 4.15 shows the process arrangement.

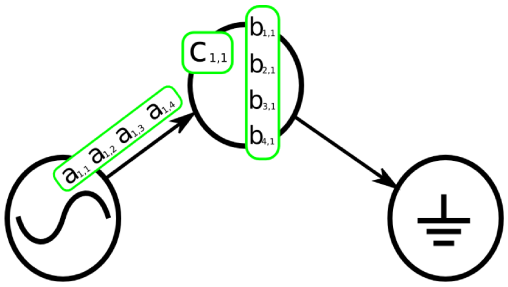


FIGURE 4.15: Serial equivalent of a pipeline.

Sideways Split: The sideways split implementation is used to test the hierarchical dependencies use for this implementation. The perceptron is split in two, which requires the stateful object to be split, and the data packets to be split as well. Figure 4.16 shows the processes arrangement after the split. The user programmer then is required to implement four signature methods. The signature methods are:

1. SplitStateFul method: The programmer receives a Serializable corresponding to the stateful object. The user should then return an array of Serializables that correspond to the new split stateful objects.
2. CoalesceStateful method: The user is given an array of Serializables, this corresponds to the list that was returned by the user in splitStateFul. Now, the basic programmer puts the array of stateful objects back together into a single unit.
3. SplitData: The user is given a packet of data. The packet of data corresponds to what is being sent out of the diffuse method and the compute method. The user should return an array of data packets that split the data given into two.
4. CoalesceData: The user is given an array of data to put the data back together into

a single packet and return that to the framework.

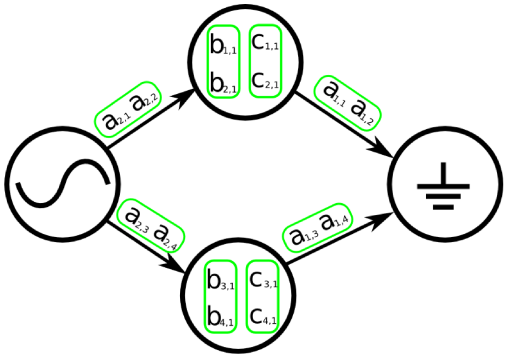


FIGURE 4.16: Sideways split using hierarchical dependencies.

The module was coded using 62 lines before adding the code that enable automatic scalability. The lines of code to enable the automatic scalability were 68. This is an increase of 82.93%.

Stage Split: The staged split is a more obvious form to automatically scale this pipeline, since the sideways implementation was done to test the performance and overhead that corresponds to the hierarchical dependencies. In this implementation, the perceptron with ten stages, would be split longways into two perceptrons where each has five stages. This automatic scalability implementation does not add lines of code on the side of the basic programmer. Figure 4.17 shows the process arrangement for this auto-scalability implementation.

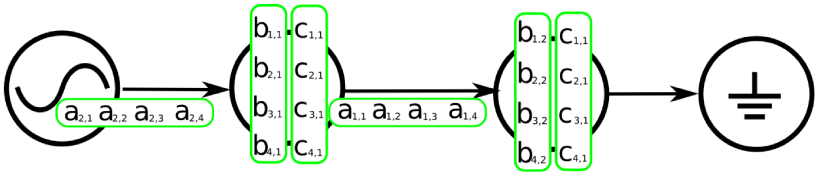


FIGURE 4.17: Staged split using dependencies without hierarchical splitting.

Results

Because both the Sideways split and the Staged split add the overhead measured on

the previous section due to the process of splitting and coalescing the perceptron, the next experiment measured only a portion of the total computation. The experiment consisted of multiplying two matrices of 2,000 by 2,000 doubles. The measurement is done in the middle of the computation to eliminate the influence of the splitting action that happens on iteration 900, and the coalescing action that happens in iteration 2,000. The measurement is therefore, the time taken to compute from iteration 1,000 to 1,500.

The measurement was taken on the Coit-grid Shared Memory, and on two servers from Coit-grid Cluster (coit-grid01and coit-grid02). The standard deviation relative to the average for the Serial, Sideways Split, and Staged Split were 1%, 4%, and 9% respectively. The specifications for the systems are given in Chapter 2. Table 4.1 shows the speedup for the experiment.

Sideways scalability does spend more CPU cycles due to the need to constantly split and coalesce the data packets when going from one perceptron that is split to a perceptron that has a trunk dependency. The shared memory environment is more sensitive to overhead also because of the proximity of the hardware components.

TABLE 4.1: Speedup using dynamic auto-scalability on Seeds.

Speedup	Serial(No Split)	Sideways	Staged
Shared Memory	1	1.7	1.8
Distributed Memory	1	1.46	1.43

In the cluster test, the extra computation incurred by the sideways scalability implementation does not have as big of a influence because it is masked by the network delay. Table 4.1 shows the speedup for the experiment on the Coit-grid Cluster system (bottom).

The staged implementation may have experience slightly lower scalability given that it misses half a computation cycle because the second stage has to wait for the data to get to it on the first data computation.

Discussion

The experiment shows the automatic scalability feature consisting of extra computation to split the data packets into smaller packets is competitive with an automatic scalability algorithm where the splitting/coalescing is not necessary. A significant cost is paid in adding the feature to existing code, with 82.93% increase in code makes the features not as attractive as first expected. This mostly means the feature would be used during optimization phases for this type of algorithms.

It should be noted that the pipeline is slightly changed from previous basic layer interfaces. The new pipeline has two optional signature methods that can be used to load initialization data to the stages. The signature methods are:

onLoadStage(): should return a Serializable stateful object.

onUnloadStage(): retuns a stateful Serializable object to the user programmer.

The hierarchical dependencies adds a negligible amount of overhead to the computations, while at the same time adding more non-functional code to the implementation.

4.12.3 Jacobi Heat Distribution Algorithm using a Hierarchical 5-point Stencil

This section shows results after using hierarchical dependencies to implement a 5-point stencil. The 5-point stencil parallel programming pattern was implemented mainly to measure programmability for the hierarchical dependencies. The algorithm used for the experiment was the Jacobi solution for the heat distribution problem[51]. The results

includes performance overhead and programmability measurements. The use of automatic scalability on the heat distribution problem increases the LOCs by 18.23% over a version without automatic scalability, and its performance overhead was 1.47% compared with a calculated ideally scaled version. At the advanced layer, data flows with hierarchical dependencies were used to enable the use of automatic scalability.

The implementation for the stencil pattern may need additional input from the basic programmer depending on the advanced user's implementation. The basic user may need to provided methods to divide the initial state (the sub-matrix sent at the beginning of computation), or when sending and receiving data during the synchronization stage.

Figure 4.18 shows a four perceptron stencil. The advanced user is responsible for allocating an arbitrary numerical identification to each perceptron and each dependency. In the example on the figure, the dependencies are numbered clockwise starting with the right dependency. Perceptron zero has output dependencies 0:0 and 0:1. The perceptron has the input dependencies 2:3 and 1:2.

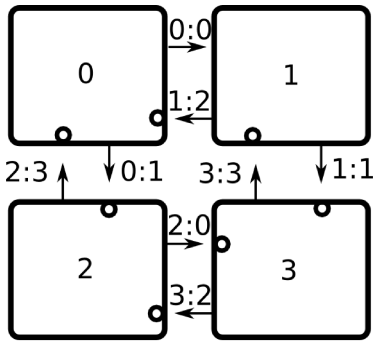


FIGURE 4.18: ID tags for dependencies and perceptrons.

The stencil pattern is shown on figure 4.19. The signature methods on the top are required in order to run the application successfully. The bottom signature methods, the

```
1.  public abstract class Stencil extends BasicLayerInterface{
2.      public abstract boolean OneIterationCompute( StencilData data);
3.      public abstract StencilData DiffuseData(int segment);
4.      public abstract void GatherData(int segment, StencilData dat);
5.      public abstract int getCellCount();
6.      public abstract StencilData[] onSplitState
7.                  (StencilData data , int level);
8.      public abstract StencilData onCoalesceState
9.                  ( StencilData[] dats, int level);
10.     public abstract Serializable[] splitData( Serializable serial);
11.     public abstract Serializable coalesceData( Serializable[] packets);
12. }
```

FIGURE 4.19: The stencil pattern required signature methods, and the extra signature methods added to the stencil pattern to provide auto-scalability. Implementing the extra methods is optional for basic function, but required to enable auto-scalability.

ones inside the rectangle, are extra methods that are needed in order to maintain the 5-point stencil general enough for many types of implementations, yet provide automatic scalability. An alternative implementation, for example, could assume the use of a 2D array of doubles. In that implementation, the advanced user would be able to completely hide auto-scalability non-functional concerns from the basic user. In the implementation shown in the figure, however, the basic user is free to use a data structured for the problem, not necessarily a 2D array.

Figure 4.20 shows graphically how the stencil splits its processing units. In this implementation, the perceptron is split into two perceptrons, and the cut is done horizontally. The advanced user is responsible to create new dependencies that will connect the two new perceptrons, and the dependencies at the side of the old perceptron would use

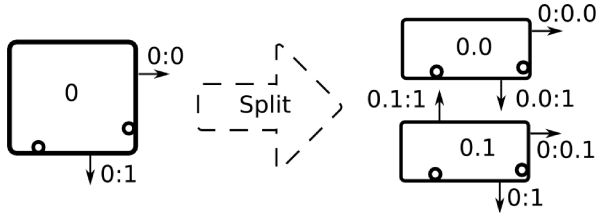


FIGURE 4.20: From left to right, a single perceptron is split into two perceptrons. The new dependencies 0.0:1 and 0.1:1 are created, and the existing dependency 0:0 is split into 0:0.0 and 0:0.1.

the hierarchical dependency feature once deployed.

Shared Memory Test: The performance test for shared memory was done on Coit-grid Shared Memory. The test was done to measure the overhead incurred by the automatic scalability feature. The total time taken to run the stencil pattern was measured. Figure 4.21 shows the results. The static results, as a whole, show the framework's performance when running with the automatic scalability feature turned off. The dynamic test shows the time it takes for the framework to complete the pattern using hierarchical dependencies to automatically scale the program. The precise procedure is as follows: four processes start running the stencil. At iteration 1,500, the perceptrons hibernate computation and return to the master node to be split into eight perceptrons. Upon allocation of the perceptrons, the computation continues until iteration 3,000 is reached. Finally, the ideal value is calculated using the results from the static measurements of four processes, and the calculated value for eight processes. The graph shows that the framework was able to in-

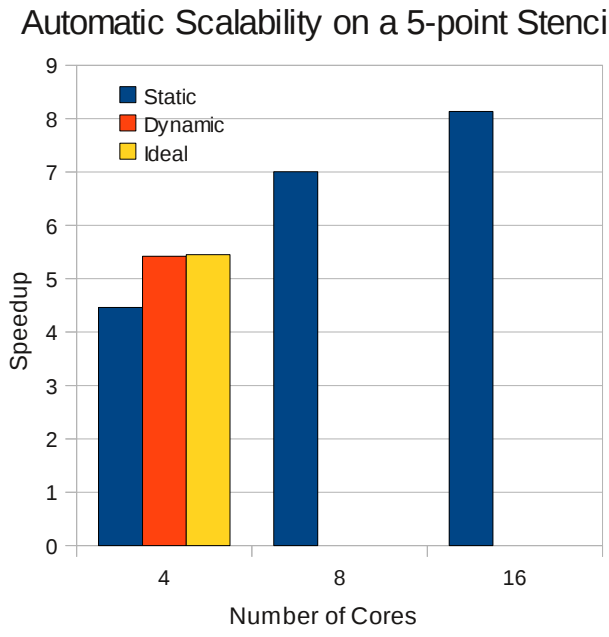


FIGURE 4.21: Performance test to measure auto-scalability's overhead.

crease performance during run time at a small performance overhead. The overhead for this test was 0.54% compared to the ideal. The standard deviation for the test was eight seconds.

A test was done using a 5,000x5,000 matrix. With that test, the results were less predictable. The static tests went down for 8 cores, and not high enough for 16 cores. This may be because of contention on the synchronized variables. The problem is performance related, and is more related to the framework as a whole. The phenomenon will be addressed in Chapter 5.

Distributed Memory: The tests done on distributed memory are not as friendly to analysis because mixing shared memory and cluster systems introduces variations in the measurments. A more practical environment for this test is the use of multiple single core computers. Figure 4.22 shows the results after running the heat distribution problem on the Coit-grid Cluster using one to four threads per core. The figure shows that the hierarchical dependencies can bring speedup in this example. However, the conditions to get speedup change with the grain-size and number of cores used. A similar test expanding from two cores to four cores shows no speedup improvement, this means the speedup potential in expanding a pattern should be measure by a load balancer before committing the parallel application to an expansion. A related parameter that can be consider is the amount of work that is estimated will be done with the greater number of cores. For example, if the parallel application were to split after 1,500 iteration from four cores to 16 cores, but there are 20,000 iterations left, at that point, the overhead can be justified.

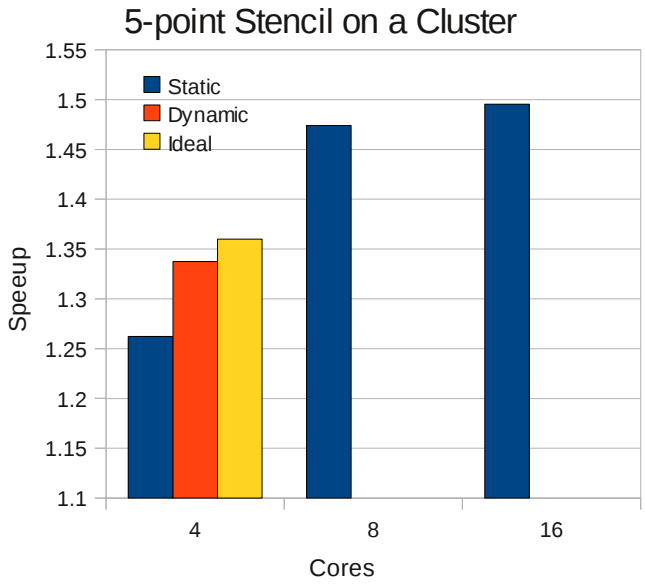


FIGURE 4.22: Performance test on Coit-grid Cluster using a mixed multi-core, multi-thread, cluster.

Another issue in cluster testing is that not all the perceptrons may hibernate. Out of four, three perceptrons may reach the consensus to hibernate, but the fourth one may miss the signal. Although this has no effect on the program's correctness, it does require extra care when benchmarking the framework. The parameters mentioned in Section 4.11 can be manipulated to ensure all perceptrons hibernate at the same time for benchmarking purposes.

4.13 Programmability

In order to enable auto-scalability in the program, the user programmer must add four signature methods in the case of the stencil pattern. For this test, the total increase in line count was 74 LOCs. The original pattern implementation was 406. After adding the extra lines, the code increased to 480. Anecdotally speaking, the extra lines of code were not a challenge. The basic function of the methods was to split either the stateful matrix or the arrays of point into two objects. How hard this extra step is dependent on the prob-

lem being implemented. The total line increase was 18.23%. Figure 4.23 shows the code created to split the send and receive routines. In splitData method, the task is to create two arrays of double (lines 5 and 6), and allocate the data from the supplied data (lines 1 and 2) to the new arrays (lines 7 through 12). The two arrays are returns as an array. coalesceData method does the same thing but in reverse.

OnSplitState and OnCoalesceState work similarly but for the stateful object, and in contrast to the previous two, the state signature methods are only called once.

```
1. public Serializable[] splitData(Serializable serial) {
2.         double[] packet = (double[]) serial;
3.         int half = packet.length / 2;
4.         int whole = packet.length;
5.         double[] one = new double[half];
6.         double[] two = new double[whole - half];
7.         for( int i = 0 ; i < half ; i++){
8.               one[i] = packet[i];
9.         }
10.        for( int i = half; i < whole; i++){
11.              two[i - half] = packet[i];
12.        }
13.        Serializable[] ans = {one, two};
14.        return ans;
15. }
16. public Serializable coalesceData(Serializable[] packets) {
17.        double[] one = (double[]) packets[0];
18.        double[] two = (double[]) packets[1];
19.        double[] united = new double[one.length + two.length];
20.        for( int i = 0 ; i < one.length; i++){
21.              united[i] = one[i];
22.        }
23.        for( int i = one.length; i < one.length + two.length; i++){
24.              united[i] = two[i - one.length];
25.        }
26.        return united;
27. }
```

FIGURE 4.23: Implemtation for split and coalesce signature methods.

4.14 Related Work

The data-flow has been used by Aldinucci et al. For Muskel's implementation to provide a back-end that interprets and executes skeletons; the approach only uses Directed Acyclic Graphs (DAGs)[57]. DryadLynq (Yu et al.) also uses a data flow DAG's to gen-

eralize the benefits of the MapReduce API [59] These projects mostly focus on skeletons, and do not address synchronous parallel algorithms. This is by design, since the main objective for the projects is to simplify development for large data sets that need simple repetitive algorithms (embarrassingly parallel) applied to them. Synchronous Data-flows exist conceptually in areas of electrical, and computer engineering as well as computer science. There are synchronous data flow languages: Lustre [60], and Signal [61] are examples of the approach. However, the languages implement the synchronous data flow concept in a very fine grain manner. Most modern literature on the concept is related to embedded systems [62]. Our data flow implementation does assume streams of information as opposed to fine-grain, single instruction connection. In this regard, the implementation is similar to the FastFlow by Aldinucci et al. [63] and StreamIt by William et al.[64]. To our knowledge, synchronous data flows has not been used as a parallel programming approach in a Grid/Heterogeneous environment as neither has the hierarchical dependency concept.

4.15 Conclusions

Adding hierarchical dependencies to the data flow technique simplifies the task of adding automatic scalability. The technique was paired with parallel patterns to address issues involving programmability. This chapter presents the conceptual and technical details behind the use of hierarchical dependencies. The chapter presents an overview of the data flow approach to parallel programming, as well as the implementation details done on Seeds to enable the feature. The chapter presents the concept of hierarchical dependencies, and its most important implementation details. The results made so far have shown that the extension to data flow programming can work, although more optimiza-

tion has to be done before the overhead is down to a level where performance improvement can be seen on both cluster systems and shared-memory systems.

CHAPTER 5: OPTIMIZING THE SEEDS FRAMEWORK

Skeletons and patterns provided a level of abstraction that allows the programmer to create parallel programs with less effort than the alternative standards of OpenMP and MPI. With skeletons/patterns, the programmer is able to get a high performance prototype in a time that is closer to the time it takes to develop the domain-specific algorithm. However, once the prototype is proven, the software is optimized to use the resources more efficiently. Improving the program so that it accomplishes its tasks in less time is the optimization that is generally sought. The program can also be optimized to use less energy. We will focus on optimization to complete the task in less time.

For the most part, the programmer gets full access to the computer resources during the computation phase in the skeleton/pattern. For example, the user can use Java Native Interface (JNI) to optimize algorithms using C language that could speed up performance on a specific hardware environment. The user could also make use of GPU libraries such as JCUDA[65] to make use of GPU computing if available on a computation node. Therefore, the programmer is less constrained in regards to access to the machine's hardware.

During the communication phase, however, the user must rely on the framework's services to reach optimum communication performance. This chapter presents two improvements done to the framework's communication infrastructure. The framework uses

the Java Serializable and Externalize interfaces to unmarshall and marshall the data packets that are returned by the programmer's implemented methods. The second improvement was eliminating the use of a dependency wrapper packet to include an asserting version number along with each data packet.

5.1 Serializable Overhead

The framework uses the Serializable interface to unmarshall and marshall the data packets before and after the data is sent over the network. The Serializable interface impacts programmability positively because the JVM will take on the job of converting the object packets into a stream of bytes. However, the Serializable interface adds several bytes of extra information that increases the size of the packet.

The Serializable's unmarshalling algorithm first writes a head message into the stream. This includes the name of the class and its version. Because the name of the class is included, the number of bytes used by the programmer when naming the class will add to the overhead. A one letter empty class will have 22 bytes of overhead. Additionally, adding primitives also adds overhead since the name of the variables are also added. A one letter name class with one integer that is also named with one letter uses 30 bytes. This is four bytes for the variable's name and four bytes for the integer's value. Adding a double adds 12 bytes to the packet. An empty array has 27 bytes of overhead. Adding more letters to each of the names adds one byte of overhead each. Additional overhead is added if the packet class extends another class, but not if it implement an interface. Additional overhead is also added if other variables inside the packet class are objects.

It is not recommended to advise the programmer against using long variable names or embedding objects inside the packet class because that can affect the code's readability.

Instead, the better approach is to allow the programmer to create readable code and data structures that are optimal for the algorithm being implemented. The efficiency issue can be resolved during the optimization phase. The Externalize interface helps with this strategy.

The Externalize interface adds 25 bytes of overhead if the object is not empty. Externalize can be helpful in optimizing the packet size because the interface allows the programmer to send primitives into the stream. There is no need to send variable's names and this helps reduce overhead. The interface lets the programmer free to use any number of letters to name the variables and to use objects inside the packet class. The rest of the marshalling process is transferred to the programmer by means of a couple of signature methods. The two signature methods are:

- void writeExternal(ObjectOutput arg0): Is used by the programmer to insert the objects variables into the provided (arg0) stream. The JVM calls the method if the object needs to be serialized.
- void readExternal(ObjectInput arg0): is called by the JVM to read the byte stream and to create the original data object.

We developed a third option that provides an interface called RawByteEncoder. The signature methods for it are:

- void toRawByteStream(ObjectOutputStream stream, Serializable obj): The programmer should cast the Serializable to the object intended. Alternatively, the user can use *instanceof* keyword to first determines which type of object it is. This can be the case if multiple objects are being used to send and receive data.

Each *instanceof* call can take from one millisecond to 100 nanoseconds (once the JVM compiles the procedure). The programmer should first sent an identifier byte before starting with the custom data. In contrast to Finalize where the signature methods are implemented for each Finalize object, the RawByteEncoder is meant to be used to marshall and unmarshall multiple packet objects.

- Serializable fromRawByteStream(byte identifier, ObjectInputStream stream): The programmer is given an identifier. The identifier is used to determine which object should be instantiated to receive the data. Then the programmer can start retrieving the custom data to create the object. The programmer then returns the received object.

The interface should be implemented on a stand-alone class. The class is then offered to the framework implementing a method called *getRawByteEncoder()*. The method is overridden from *BasicLayerInterface*. This is the parent abstract class to all skeleton and pattern interfaces. Seeds checks for the existence of a *RawByteEncoder* at its communication module (*MultiModePipe*). If a packet arrives with an identifier byte that is not in the list of reserver numbers, the framework checks if a *RawByteEncoder* is available. If so, the class is used to interpret the stream. From the perspective of the programmer, the interfaces behaves the same as Externalized.

RawByteEncoder eliminates the need for the initial 25 byte header needed by Externalizable. This is replaced with a 1 byte header that identifies the object to the framework. Instead of encoding the class name, the programmer can create a single byte number that is used to identify the class when it is coming through the pipe. The version

number is not necessary, since the purpose of it is to identify an older version of a class when this class has been stored in a hard drive for a long time. That requirement does not apply to a parallel application. In Seeds, the packet class is not expected to advance in version during the program's runtime. The protocol can handle 253 different Java objects. This is the available values for a byte minus three reserved values used to control the connection for tasks such as closing the connection or hibernating a dependency. Because the byte was already used to control the communication packet, this byte is also part of the Serialized and Externalized communication options within Seeds. Figure 5.1 shows a comparison packet between the three protocols. The data transfer in this example is a 30 integer array. It can be noticed, that the disadvantages of Serialization are minimized if the programmer uses an array with a large number of elements.

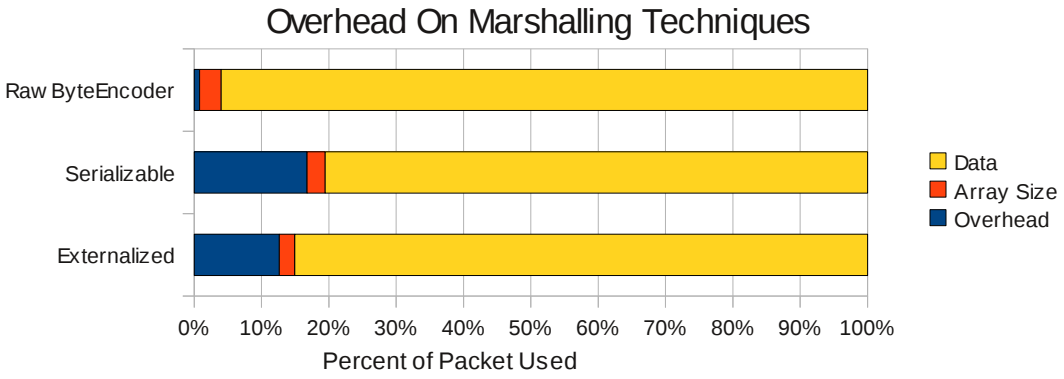


FIGURE 5.1: Overhead on marshalling techniques.

5.1.1 Results

The improvement was added to the framework's communication module. A workpool skeleton was used to test the optimization. The algorithm used was converting an RGB picture to gray scale. Figure 5.2 shows the 5000x5000 picture used for the experiment.

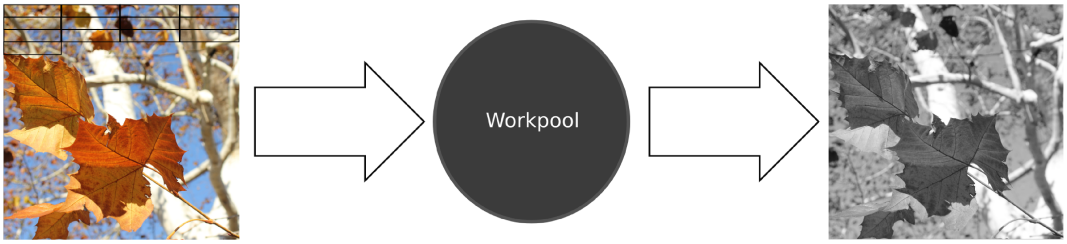


FIGURE 5.2: Parallel grays scale algorithm using a workpool with varying data packet size.

The algorithm was implemented so that a varying packet size was possible. Changing the packet size helps to show when the use of Externalize and RawByteEncoder interface would be unnecessary because Serializable would be close to them in performance. The test changed the packet size from ten integers per data packet to 5,000 integers. Figure 5.3 shows the results from the test. At 2,500 integers, the performance starts to be close enough to Externalizable and RawByteEncoder. In general, the smaller packet size can facilitate greater scalability if increasing the problem's size is not possible. At ten integers per packet, all three protocols performed worst. A good option for maximum scalability and low time to completion is using *RawByteEncoder* with 250 integers on the data packet.

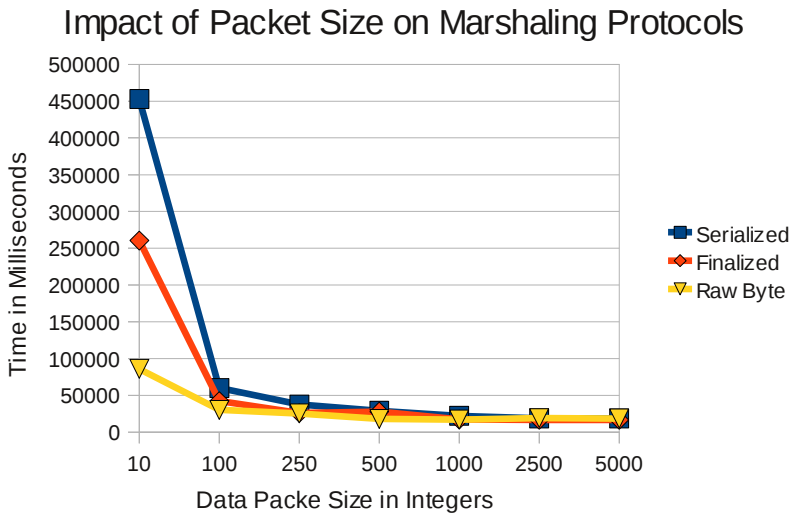


FIGURE 5.3: Test performed to measure the effects of packet size due to serialization on the Seeds framework.

5.1.2 Programmability

The gray scale serial algorithm can be coded with 31 lines of code. The Serializable parallel version using Seeds was done with 106 lines. Adding the marshalling and unmarshalling to the code added from ten (Externalizable) to twenty (RawByteEncoder) lines of code. The extra effort on the part of the programmer has to be incurred for each object that needs to travel over the network. Even though RawByteEncoder and Externalizable are very similar in implementation, the lines of code for RawByteEncoder are more because Externalizable is implemented as part of the data packet object; RawByteEncoder on the other hand, is implemented as a standalone class that is then returned to the framework.

The improvement is significant, since it can encourage the domain-specific programmer to develop finer grain algorithms, and the algorithms can in turn be more easily scaled by the framework if the resources allow it. Another advantage from interfaces is that they allow for the separation of the implementation code from the optimization code,

which leaves the original domain-specific code more readable.

5.2 Hierarchical Dependency Optimization

The hierarchical dependencies initial implementation transfer a version integer along with the user's data. It also wrapped the user's data in order to add the version to it. The implementation helped validate the hibernation algorithm, and the reconnection algorithm. Using the version integer, the algorithm was able to verify the programmer's algorithm resumed computation exactly where it left off. On the down side, the wrapper DependencyPacket class added extra overhead to the data packets when the information flows across computers in a cluster or Grid.

In order to improve performance, the version integer and the wrapper *DependencyPacket* object were taken out. The matrix multiplication algorithm was run again using the improved version. The characteristics were left the same as the experiment shown in Section 4.12.2. Table 5.1 shows the results from this test. The results count 500 iterations done towards the end of the computation. This provides a measurement that does not include the overhead due to the perceptrons stopping the computation and splitting, thereby allowing accurate measurement of the communication costs created by the hierarchical dependency.

TABLE 5.1: Speedup comparison between the first implementation and the implementation presented in this section.

	NoSplit	Staged Split	Sideways Split
First Implementation	1	1.43	1.46
No Version Assert	1	1.5	1.6

The results were positive for the split version, but the results were the same for the serial, no split version. The test was performed fifty times to increase confidence in the results. The stage split gain 4.05% and sideways split gained 7.84%. The gain is mainly due to eliminating the wrapper dependency packet. The RawByteEncoder interface was used for this test as well.

5.3 Framework Scalability and Validation

This section presents a general assessment of the scalability of Seeds. For this experiment, previously mentioned algorithms were run with a high number of cores to measure how the framework handled the load. Grain size was varied for the different tests. Both Coit-grid Shared Memory and the Coit-grid Cluster were used for all tests.

The Monte Carlo calculation of π was run. The test presented in Chapter 2 was done with 100,000 jobs each of them computing 10,000 numbers to approximate π. Figure 5.4 shows the repetition of this test. Each run was repeated ten times with an average standard deviation of 3.8 seconds for the whole set. Figure 5.5 shows the same test with the same number of jobs but the number of points computed (grainsize) set to 100,000. Each measurement was taken only once for this test. The test should provided an assessment of the framework scalability features. A higher grain size is needed to get higher speedup from the framework. At the advanced layer, the workpool implementation shown here performs a load balancing algorithm (not implemented for the test in Chapter 2) that distributes the work heterogeneously to accommodate for the difference in processor speed. Coit-grid Shared Memory runs its processors at a lower speed than Coit-grid Cluster.

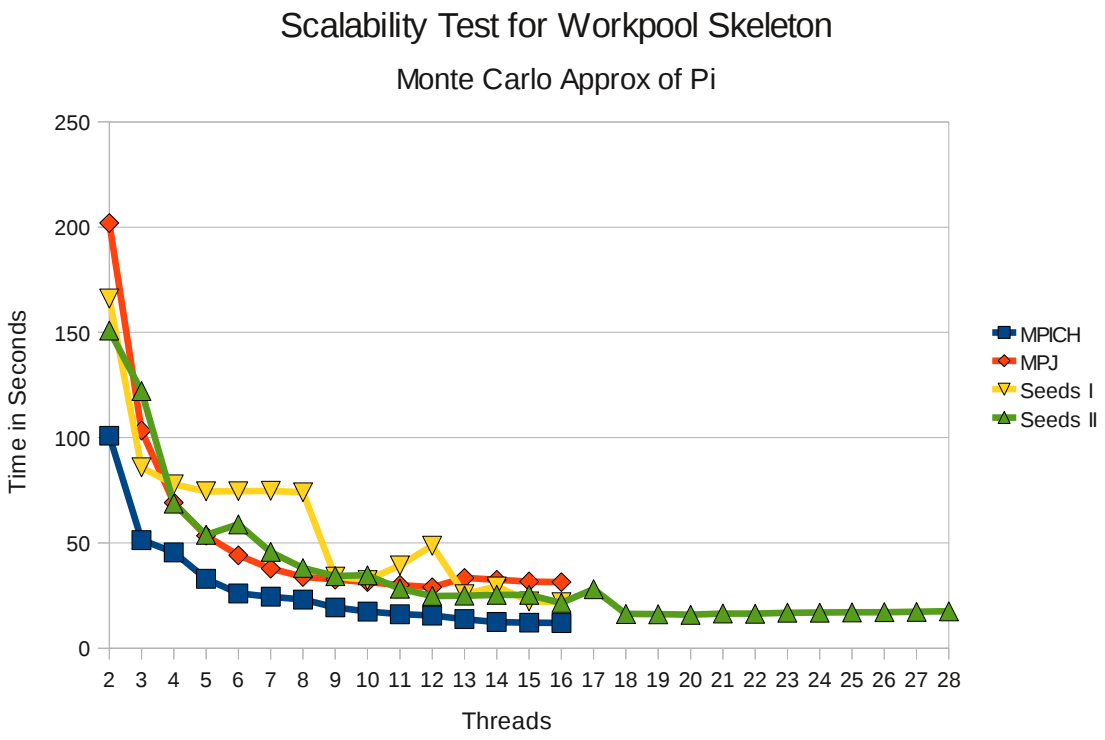


FIGURE 5.4: Seeds running with RawByteEncoder and a load balancing algorithm to adapt to a heterogeneous environment.

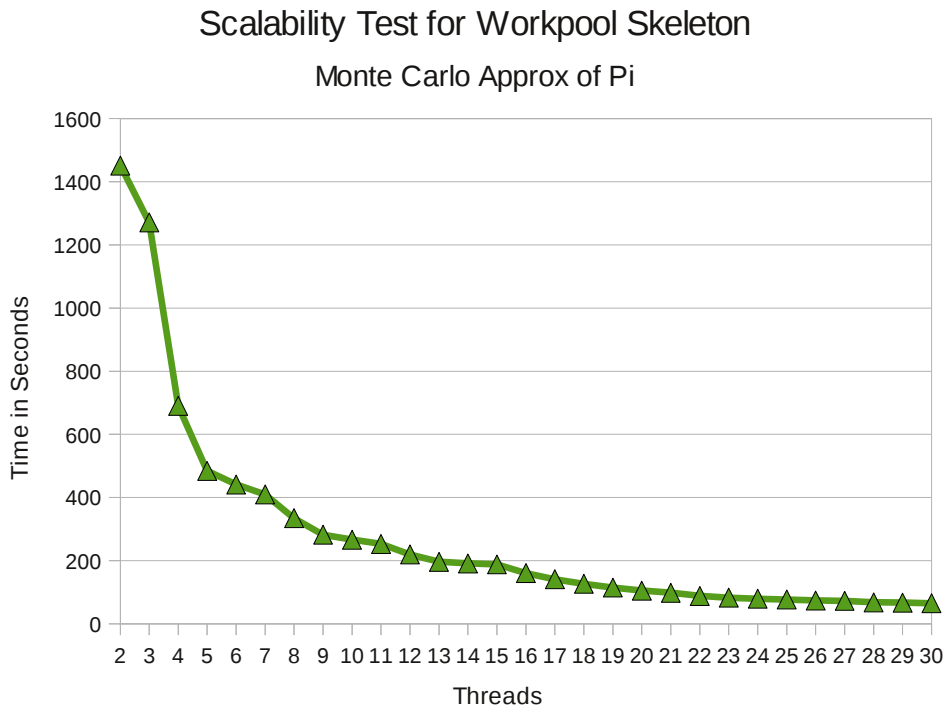


FIGURE 5.5: Scalability test with increased grain size for the Mote Carlo approximation of π algorithm.

Similarly, a scalability test was done to assess the framework with respect to more coupled patterns. The heat distribution algorithm presented in Chapter 3 was run using both Coit-grid Shared Memory and Coit-grid Cluster. The results shown in Figure 5.6 are from computing the algorithm for 3,000 iterations on a 1,000x1,000 matrix. It can be seen that, like the previous test, the framework cannot scale for this small job. Figure 5.7 shows the same test, this time with a 3,000x3,000 matrix. The increase in work from 1,000x1,000 to 3,000x3,000 is nine times. The figure shows how Seeds can scale the application when the grain size allows for it. At 32 threads, Seeds achieves a speedup of 25. Each of this tests was performed once. The stencil pattern used here is not using the RawByteEncoder interface.

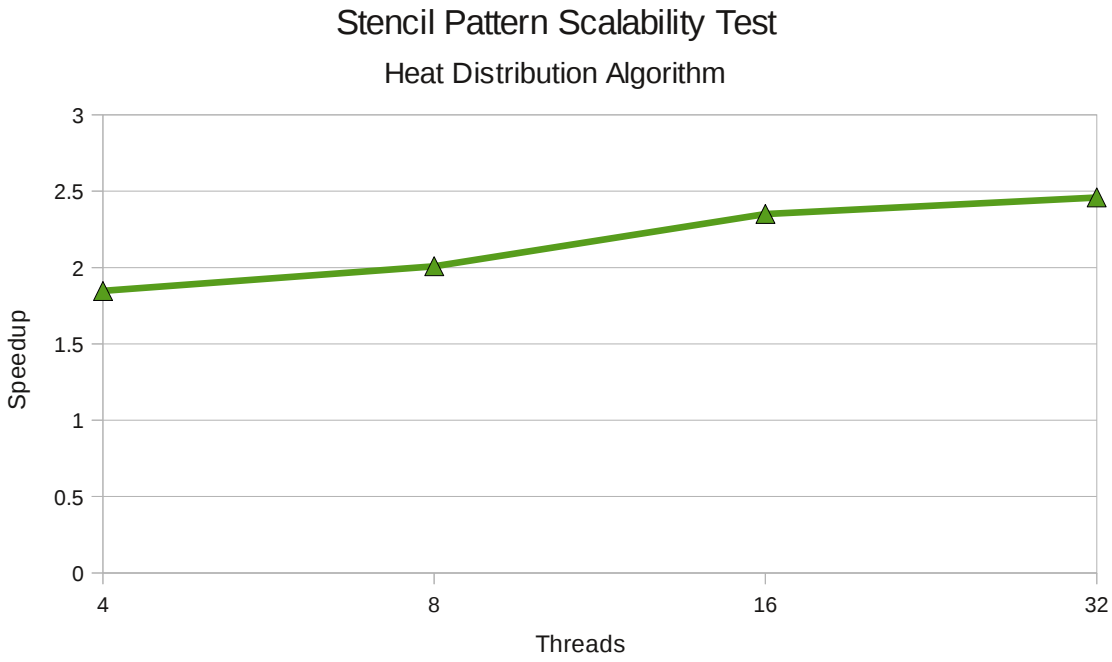


FIGURE 5.6: Heat distribution algorithm run with a static data flow configuration working on a 1,000x1,000 matrix.

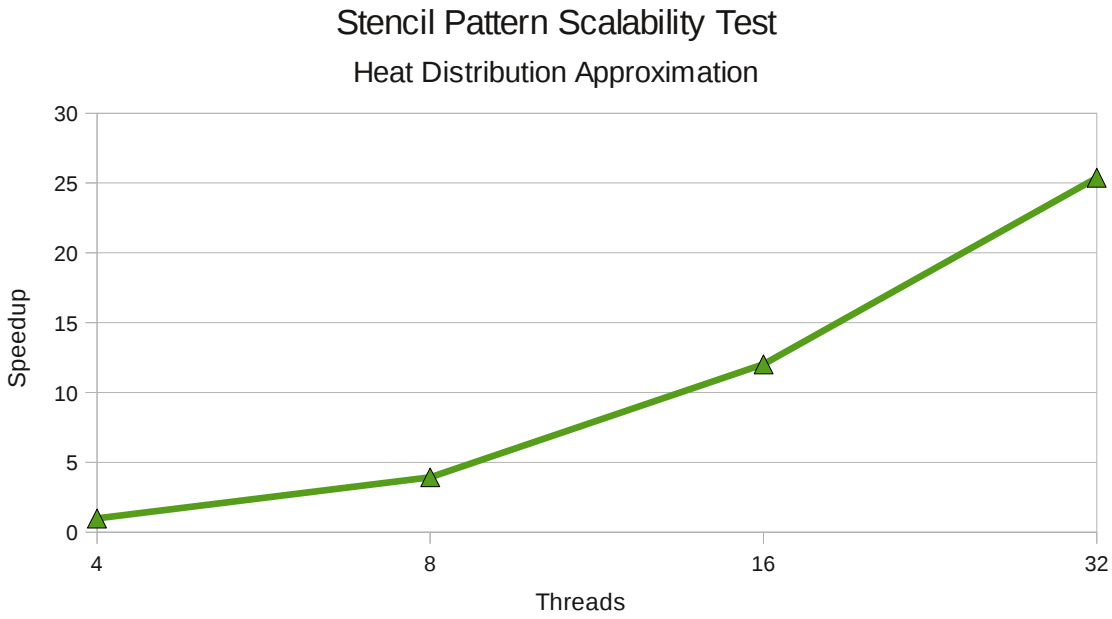


FIGURE 5.7: Heat distribution algorithm run with a static data flow configuration. The pattern worked on a 3,000x3,000 matrix.

5.4 Conclusions

This chapter addresses efficiency issues in the Seeds framework. Although skeletons/patterns can provide great help to programmers, the programming technique can also isolate the programmer from the necessary insight needed to optimize an algorithm once the initial prototype has been implemented. We conclude the programmer does get enough access to the hardware resources to optimize algorithms within the compute() methods. The programmer has to rely on the framework to provide optimum communication. The use of Serialization interface is helpful to prototype a parallel algorithm or if the algorithm uses large data packets. The interface should be replaced with RawByteEncoder interface to lower the packet overhead and increase the algorithm's scalability.

The improvements shown in this chapter work independent of the algorithm implemented by the domain-specific programmer. The use of skeleton/patterns with the extensions explored in this dissertation can help separate the programmer from the complex hardware environment. There are areas of research that can be explored to make these type of frameworks more robust. The next chapter mentions some research topics that are left as future work.

CHAPTER 6: FUTURE WORK

This chapter presents future work with regards to the Seeds framework, Pattern Operators, and Hierarchical Dependencies. The seeds framework can be modified to increase performance to make it useful in a production environment. The pattern operators can be used to add behavioral skeletons to add more features to the framework. The topic that contains most potential and areas of research is the hierarchical dependencies. They can be modified to allow for redundant perceptrons and to provide automatic ghost zone perceptrons. Scheduling and load balancing algorithm can be re-analyzed with hierarchical dependencies in mind, since the hierarchical dependencies affect multiple aspects of scheduling and load balancing that were considered constant by previous literature.

6.1 The Seeds Framework

The Seeds framework is consider “the laboratory” onto which new Grid/cloud/multi-core ideas can be implemented. Any modification to the current framework would not be of academic significance. A better improvement for the framework would be to reimplement it using a stream-based language. The research would include selecting a relatively main-stream language and porting the code to that language. The goal would be to increase the framework's performance without going backwards on the programmability features that have been developed by previous skeleton/pattern projects, as well as the ones presented in this dissertation.

6.2 Pattern Operators

An idea by Gomez et al. that can be useful to port from the Problem Solving Environment (PSE) into skeleton/pattern frameworks is to use pattern operators to add a behavioral patterns to a skeleton/pattern parallel application[58]. Behavioral patterns are patterns that are created based on their behavior, and not based on their communication patterns as is the case for the skeleton/patterns we study in this dissertations. Examples of behavioral patterns include the use of check-pointing patterns. The check-pointing pattern is used to route a partially computed stateful data onto some backup server. The check point is reloaded after a failure event. Another example is the use of visualization clients. This pattern can route filtered (optimized to be displayed on the screen) data on to a client computer where a programmer can see intermediate computation states. Similarly, a steering pattern can be used to send interactive actions from the user to "steer" a computation. These patterns can be added to the Seeds framework using the pattern loading mechanism presented in Chapter 3. The study on such endeavors would be mostly about its impact on programmability, and not so much on performance.

6.3 Hierarchical Dependencies

The three most promising research ideas that can be pursued further with respect to hierarchical dependencies are: automatic redundant processing perceptrons, automatic ghost zoning perceptrons, and researching how hierarchical dependencies can affect scheduler and load balancers.

6.3.1 Automatic Redundant Perceptrons

Automatic redundant perceptrons can be provided to the advanced user using the hierarchical dependency technique without requiring to get knowledge about what the final

application will do. Figure 6.1 shows how redundant perceptrons can be accomplished for this programming approach. In the figure, process 2 is being replicated, the output emitted by process 1 is sent to both of process 2 instantiations. The message passing that is needed to replicate the process can be done by the framework that implements the packet process features.

By implementing the redundant process connection feature into the hierarchical dependency connection, the framework can lower redundant processing from the advanced layer to the expert layer. If implemented, the advanced user can specify the need for a redundant perceptron during launch, or the framework can decide to deploy redundant perceptrons based on the environment's performance and chance for failure. But the real benefit of the feature, other than failure prevention, is its impact in creating ghost zone perceptrons.

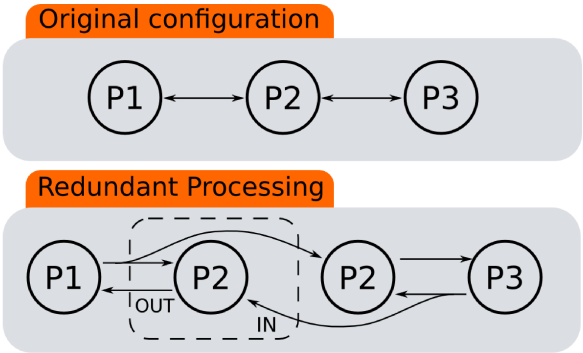


FIGURE 6.1: Message passing for data-flow redundant processing.

6.3.2 Automatic Ghost Zoning

Figure 6.2 shows graphically how ghost zoning can be accomplished using the data-flow approach. In this case, the example focuses on processes P2 and P3 and the ghost perceptron P3R. P2 sends its output to P3 and P3R. P3R uses the data to compute data closer to P4. As in redundant processing, the ghost zone technique can be added to the

data-flow abstraction, which provides the benefits of ghost-zoning to a parallel program independent of what the parallel program does. The implementation should be added to the expert layer, which should provide the capability to the advanced layer as a service.

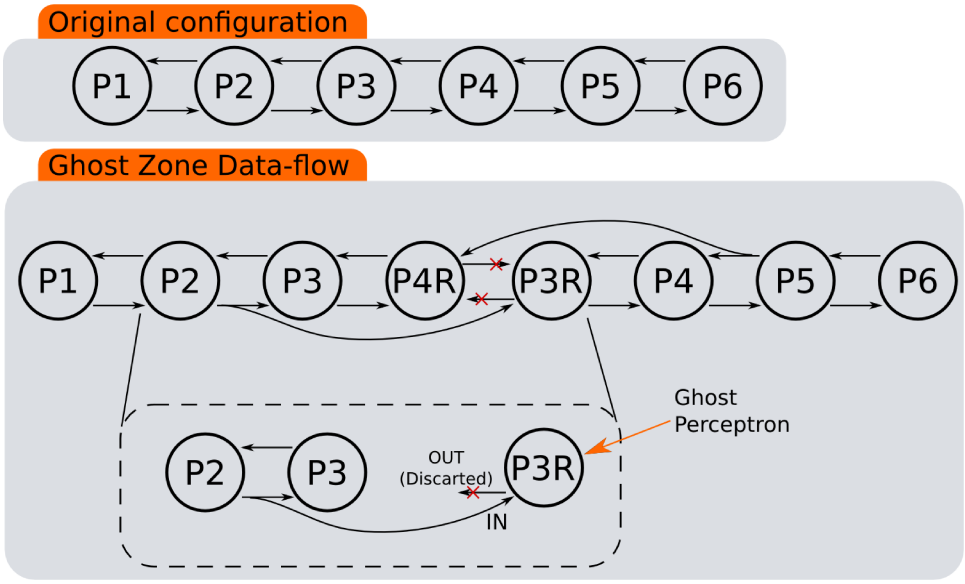


FIGURE 6.2: Ghost zoning for data-flow.

Technical Challenges

The implementation of the automatic ghost zoning for hierarchical dependencies only would cover one of the dependency connections. That is, we would assume that all other dependencies for the perceptron do not need a ghost zone. This can happen when two Grid nodes in a lambda Grid compute the same parallel program[41]. But a likelier scenario in the future is having two hosts with hundreds of cores each. In that case, the LAN connection would be orders of magnitude slower than the connection between the cores. The use of ghost zones for tightly coupled programs can be beneficial.

The way in which a pattern splits will also come into play, since the split featured would be used to create several slivers of perceptrons, each with its own version, and the slivers would have to be perpendicular to the high latency connection in the cluster. This

is necessary because ghost zoning needs to allow the version on each side of the high latency connection to fall out of sync. Because the sliver perceptron that is closest to the local computation core has the same version as the neighbor processes, the framework does not modify the algorithm's behavior. Once the sliver perceptron that is next to the neighbor process falls out of sync, it simply stops computing waiting for the version number on its side update. The behavior creates waves of streamed data that resembles a Go-Back-N TCP window[66], which are, on the average, faster than a single sliver of ghost data. Refer to Latency Hiding by Redundant Processing algorithm for more on the specific implantation of a ghost zone with a large ghost area[41]. The extension mentioned here would add the abstraction necessary to make it available to the advanced and basic users.

The main advantage behind automatic redundant perceptrons and automatic redundant ghost zone perceptrons is that the framework would then have more tools to deploy to adapt the domain-specific programmer's application to a changing environment, cloud, Grid or multi-core.

The research into these areas is validated by implementing algorithms that are different in the pattern they use. Performance should be measure to ensure the overhead added by the feature is not prohibitive. From the programmability stand point, we can compare this using similar methods as those presented in Chapter 4. The expected result is to have a significant reduction in lines of code for a comparable MPI implementation that has ghost-zoning int it.

6.3.3 Schedulers and Load Balancers

The use of Hierarchical Dependencies can also have a big impact on scheduler and

load balancing algorithms. A study of schedulers and load balancing algorithms can be done with the intent to extend these algorithms using the capabilities presented in Chapter 4. The main benefits include being able to schedule a program with the minimum number of processors needed. Once deployed, the parallel application can expand to use more resources. In contrast, current schedulers must deploy a parallel program when the necessary static amount of resources is available. Load balancing usually refers to balancing a specific, known algorithm. With hierarchical dependencies, theoretical nomenclature used in load balancing can be implemented as a module in the framework. That would allow researches to directly test load balancing theory on patterns.

6.4 Conclusions

This chapter presents a few of the most compelling subjects that can be researched starting from the literature presented in this dissertation. The Seeds framework can be reimplemented with the objective to improve performance. More operators can be added to the framework to improve programmability. Adding automatic ghost zone perceptrons and automatic redundant perceptrons to the hierarchical dependency concept can make the approach even more attractive to domain-specific programmers by adapting their application to more heterogeneous environments with little extra effort, or by postponing the optimization by separating the features from the main code using signature methods. Also, the impact of hierarchical dependencies on schedulers and load balancers can be the subject of further research.

CHAPTER 7: CONCLUSION

There is more demand every year for more approaches to developing parallel applications. The Grid, the cloud, and multi-core environment have made parallel programming main stream. The challenges that confronted cluster computing researches in the past are still with us today. We have mentioned several projects that have tackle the programmability issue while addressing performance.

We then set out to create a framework that extends onto the programmability and performance approaches presented by previous literature. The Seeds framework adapts to a changing environment where processors can come on-line and off-line during the scheduling and during runtime (not due to failure). The framework was also designed to adapt to heterogeneous networks such as NAT networks, and UPNP routers. With the features added to Seeds, we moved on to implement skeleton/pattern based development technique and to extend it advantages while reducing its disadvantages.

The pattern operators were develop to add more usability to existing patterns. More usability from existing patterns also mean the programmer does not have to develop new patterns using lower level tools. The pattern adder operator was used to code an example with 27.31% less non-functional code than a similar implementation in MPJ, and its programmability index is 13.5% compared to MPJ's 9.85%. The overhead for an empty pattern with low communication was 15%.

The idea of hierarchical dependencies is proposed. The technique should allow the advanced programmer enough versatility to create patterns and skeletons that can automatically scale. We showed how the technique can be used to create auto-scaling pipeline skeletons and how a stencil pattern can be develop so that it can also auto-scale. The technique showed an overhead of 6% when running with split dependencies on shared memory. The overhead on a cluster environment was masked by the network delay. Hierarchical dependencies showed a 18.23% increase in non-functional code when the feature was added to a 5-point stencil implementation.

Of all the work, the hierarchical dependency technique seem to hold most potential, and therefore more future work. The technique could be adapted to support automatic redundant processes, which can help prevent failure. The technique could also be adapted to automatically handle ghost zones, which can further help some types of patterns.

Skeletons and patterns, together with the extensions discussed in this dissertation, can be used to create an extra layer of abstraction between the domain specific programmer and the hardware resources. We have shown the extra layer can allow the expert and advanced programmer to resolve non-functional concerns, which make the basic user more productive. Ultimately, this programming approach could create an interface that allows the programmer to interact with parallel resources in a way similar to how a programmer today uses a database language (such as SQL) to access a program's data.

REFERENCES

[1] I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations,” *Int. J. High Perform. Comput. Appl.*, vol. 15, no. 3, pp. 200-222, 2001.

[2] E. Schnetter et al., “Cactus Framework: Black Holes to Gamma Ray Bursts,” *Petascale Computing: Algorithms and Applications, chapter 24*, Jul. 2007.

[3] G. Allen, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen, “Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus,” *Supercomputing, ACM/IEEE 2001 Conference*, p. 52, 2001.

[4] “Home - TeraGrid 09,” http://www.teragrid.org/tg09/, 13 Sep. 2009.

[5] R. Buyya, D. Abramson, and S. Venugopal, “The Grid Economy,” *Proceedings of the IEEE*, vol. 93, no. 3, pp. 698-714, 2005.

[6] R. Lämmel, “Google's MapReduce programming model -- Revisited,” *Science of Computer Programming*, vol. 70, no. 1, pp. 1-30, Jan. 2008.

[7] J. Dean and S. Ghemawat, “MapReduce,” *Communications of the ACM*, vol. 51, no. 1, p. 107, 2008.

[8] M. Leyton, “Advanced Features for algorithmic skeleton programming,” Ph.D dissertation, Universite de Nice-Sophia Antipolis, Oct. 2008.

[9] M. Aldinucci, M. Danelutto, and P. Teti, “An advanced environment supporting structured parallel programming in Java,” *Future Generation Computer Systems*, vol. 19, no. 5, pp. 611-626, 2003.

[10] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, “A library of constructive skeletons for sequential style of parallel programming,” *Proceedings of the 1st international conference on Scalable information systems - InfoScale '06*, p. 13, 2006.

[11] D. Goswami, “From Design Patterns to Parallel Architectural Skeletons,” *Journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 669-695, 2002.

[12] R. Hempel, "The MPI Standard for Message Passing," in *Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II: Networking and Tools*, pp. 247-252, 1994.

[13] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46-55, 1998.

[14] S. MacDonald, K. Tan, J. Schaeffer, and D. Szafron, "Deferring design pattern decisions and automating structural pattern changes using a design-pattern-based programming system," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 3, pp. 1-49, 2009.

[15] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389-406, 2004.

[16] S. Siu, D. De Simone, D. Goswami, and A. Singh, "Design patterns for parallel programming," *Proceeding of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 230-240, 1996.

[17] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. Santa Clara, CA: NVIDIA, 2007.

[18] "TBB Home," http://www.threadingbuildingblocks.org/, April 2010.

[19] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan, "Pattern-Based Parallel Programming," *Proceedings of the 2002 International Conference on Parallel Processing*, p. 257, 2002.

[20] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications," *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pp. 97-105, 2001.

[21] M. Aldinucci et al., "Components for High-Performance Grid Programming in Grid.IT," *Component Models and Systems for Grid Applications*, pp. 19-38, 2005.

[22] M. Aldinucci and M. Danelutto, "Skeleton-based parallel programming: Functional and parallel semantics in a single shot," *Computer Languages, Systems & Structures*, vol. 33, no. 3, pp. 179-192, 2007.

[23] M. E. F. Maia, P. H. M. Maia, N. C. Mendonca, and R. M. C. Andrade, "An Aspect-Oriented Programming Model for Bag-of-Tasks Grid Applications," *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pp. 789-794, 2007.

[24] R. C. Gonalves and L. Sobral, "Pluggable parallelisation," *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pp. 11-20, 2009.

[25] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203-231, May. 2006.

[26] A. L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel, "Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers," in *Parallel Processing Symposium, International*, p. 474, 1997.

[27] V. W. Freeh, "Dynamically Controlling False Sharing in Distributed Shared Memory," *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, p. 403, 1996.

[28] A. Wollrath, R. Riggs, and J. Waldo, "A distributed object model for the java system," in *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, pp. 17-17, 1996.

[29] J. Tan, D. Abramson, and C. Enticott, "REMUS: A Rerouting and Multiplexing System for Grid Connectivity Across Firewalls," *Journal of Grid Computing*, vol. 7, no. 1, pp. 25-50, Mar. 2009.

[30] Y. Tanaka, M. Sato, M. Hirano, H. Nakada, and S. Sekiguchi, "Performance evaluation of a firewall-compliant Globus-based wide-area cluster system," *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pp. 121-128, 2000.

[31] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal, "Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems," *High-Performance Distributed Computing, International Symposium*, pp. 97-106, 2004.

[32] N. Karonis, “MPICH-G2: A Grid-enabled implementation of the Message Passing Interface,” *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551-563, 2003.

[33] H. Nakada and S. Matsuoka, “A Java-based programming environment for hierarchical Grid: Jojo,” *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium*, pp. 51-58, 2004.

[34] J. Maassen and H. E. Bal, “Smartsockets,” *Proceedings of the 16th international symposium on high performance distributed computing - HPDC '07*, p. 1, 2007.

[35] M. Bornemann, R. V. van Nieuwpoort, and T. Kielmann, “MPJ/Ibis: A Flexible and Efficient Message Passing Platform for Java,” *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2005, pp. 217-224.

[36] I. Foster, “Globus Toolkit Version 4: Software for Service-Oriented Systems,” *Network and Parallel Computing*, 2005, pp. 2-13.

[37] “GridMPI,” *http://www.gridmpi.org/index.jsp*, 13 Sep. 2009.

[38] “JXTA™,” https://jxta.dev.java.net/, Oct-2009.

[39] S. Genaud and C. Rattanapoka, “A Peer-to-Peer Framework for Robust Execution of Message Passing Parallel Programs on Grids,” *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2005, pp. 276-284.

[40] K. Shudo, Y. Tanaka, and S. Sekiguchi, “P3: P2P-based middleware enabling transfer and aggregation of computational resources,” *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 1, pp. 259-266 Vol. 1, 2005.

[41] J. Villalobos and B. Wilkinson, “Latency hiding by redundant processing: a technique for grid-enabled, iterative, synchronous parallel programs,” *Proceedings of the 15th ACM Mardi Gras conference*, 2008.

[42] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, “MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging,” *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 25, 2003.

[43] “Ganymed SSH-2 for Java,” http://www.ganymed.ethz.ch/ssh2/, Mar. 2010.

[44] J. Villalobos and B. Wilkinson, "Skeleton/Pattern Programming with an Adder Operator for Grid and Cloud Platforms," *Proceedings of the 2010 International Conference on Grid Computing & Applications, GCA 2010*, Jul. 2010.

[45] J. Villalobos, "Parallel Grid Application Framework," http://coit-grid01.uncc.edu/seeds/tutorials.php, Jan. 2011.

[46] G. Antoniu, L. Cudennec, M. Jan, and M. Duigou, "Performance scalability of the JXTA P2P framework," *Parallel and Distributed Processing Symposium, International*, p. 109, 2007.

[47] G. Antoniu, P. Hatcher, M. Jan, and D. Noblet, "Performance evaluation of JXTA communication layers," *Cluster Computing and the Grid, IEEE International Symposium*, vol. 1, pp. 251-258, 2005.

[48] J. Verstrynge, *Practical JXTA, Cracking the P2P puzzle*, Lulu Enterprises, Inc, 2008.

[49] D. Henty, "Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling," *Supercomputing, ACM/IEEE 2000 Conference*, p. 10, 2000.

[50] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. D. Groot, and E. Lear, "Address Allocation for Private Internets," *RFC 1918*, Feb. 1996.

[51] B. Wilkinson, *Parallel programming : techniques and applications using networked workstations and parallel computers*, 1st ed. Delhi: Pearson Education Asia, 2002.

[52] "MPICH2," http://www.mcs.anl.gov/research/projects/mpich2, Oct. 2009.

[53] M. Baker, B. Carpenter, and A. Shafi, "MPJ Express: Towards Thread Safe Java HPC," *Cluster Computing, 2006 IEEE International Conference*, pp. 1-10, 2006.

[54] G. L. Taboada, J. Touriño, and R. Doallo, "Java Fast Sockets: Enabling high-speed Java communications on high performance clusters," *Computer Communications*, vol. 31, no. 17, pp. 4049-4059, Nov. 2008.

[55] U. Trottenberg, C. Oosterlee, and A. Schuller, *Multigrid*. California: Elsevier Academic Press, 2001.

[56] R. Haberman, *Applied Partial Differential Equations*, Fourth. Pearson Prentice Hall, 2004.

[57] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi, "Muskel: an Expandable Skeleton Environment," *Scalable Computing: Practice and Experience*, pp. 325-341, 2008.

[58] M. C. Gomes, O. F. Rana, and J. C. Cunha, "Pattern operators for grid environments," *Sci. Program.*, vol. 11, no. 3, pp. 237-261, 2003.

[59] Y. Yu et al., "DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language," *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pp. 1-14, 2008.

[60] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: a declarative language for real-time programming," *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 178-188, 1987.

[61] P. Amagbégnon, L. Besnard, and P. L. Guernic, "Implementation of the data-flow synchronous language SIGNAL," *SIGPLAN Not.*, vol. 30, no. 6, pp. 163-173, 1995.

[62] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64-83, Jan. 2003.

[63] M. Aldinucci, M. Meneghin, and M. Torquati, "Efficient Smith-Waterman on Multicore with FastFlow," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, pp. 195-199, 2010.

[64] William Thies, Michal Karczmarek, and Saman Amarasinghe, "StreamIt: A Language for Streaming Applications," *Proc. 11th Int'l Conf. Compiler Construction, LNCS 2304, Springer-Verlag*, pp. 179-196, 2002.

[65] Y. Yan, M. Grossman, and V. Sarkar, "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA," *Lecture Notes in Computer Science*, vol. 5704, p. 887, 2009.

[66] J. Kurose and K. Ross, *Computer Networks*. Addison Wesley, 2010.