A MODEL FOR COMPOSIBLE AND EXTENSIBLE PARALLEL ARCHITECTURAL SKELETONS

Mohammad Mursalin Akon

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

December 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-04438-1
Our file Notre référence
ISBN: 0-494-04438-1

Abstract

A Model for Composible and Extensible Parallel Architectural Skeletons

Mohammad Mursalin Akon

Application of pattern-based approaches to parallel programming is an active area of research today. The main objective of pattern-based approaches to parallel programming is to facilitate the reuse of frequently occurring structures for parallelism whereby a user supplies mostly the application specific code-components and the programming environment generates most of the code for parallelization. Parallel Architectural Skeleton (PAS) is such a pattern-based parallel programming model and environment. The PAS model provides a generic way of describing the architectural/structural aspects of patterns in message-passing parallel computing. Application development using PAS is hierarchical, similar to conventional parallel programming using MPI, however with the added benefit of re-usability and high level patterns. Like most other pattern-based parallel programming models, the benefits of PAS were offset by some of its drawbacks such as difficulty in: (1) extending PAS and (2) skeleton composition. *SuperPAS* is an extension of PAS that addresses these issues. *SuperPAS* provides a skeleton description language to describe a skeleton in a generic way. Using SuperPAS, a skeleton developer can extend PAS by adding new skeletons to the skeleton repository (i.e., extensibility). SuperPAS also makes the PAS system more flexible by defining composition of skeletons. In this thesis, we describe the model and description language for SuperPAS and elaborate its use through examples.

Acknowledgments

I would like to thank my supervisor, Professor Dhrubajyoti Goswami, for being my supervisor. His guidance, support, and encouragement throughout my studies in Concordia made him more than a guru to me. I am also thankful to Professor Hon Fung Li for being my unofficial supervisor. Without his mind-blowing ideas, thinking power, patience and time, I would not be able to be confident in writing this thesis.

I would like to acknowledge students of the course *COMP628: Computer Systems Design* of Spring 2004 session for their participation in usability testing. Their comments about the SuperPAS system were beneficial.

My special thanks go to my parents, my brother and my wife for their moral support throughout my research. Finally, I am grateful to the great Almighty for everything in my life. All praise is for the Him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With time, computer hardware is getting inexpensive and faster. At the same time, scientists are investigating increasingly complex problems with finer level of detail, requiring larger computing power, sophisticated algorithms and cutting-edge software. Research in High Performance Computing (HPC) is exploring different aspects of available and foreseeable technology to realize those complex problems.

Parallel application design and development is a major area of focus in the domain of high performance computing. With the advent of fast networks of workstations and PCs, it is now becoming increasingly possible to develop high-performance parallel applications using the combined computing powers of these networked-resources at a reasonable price-performance ratio. In practice, it can be found that most of the top 500 supercomputing machines [1] are built using off-the-shelf processors. Contrast this to the situation, a few years back parallel computing was confined only to special-purpose parallel computers, each priced high enough to be affordable only by major research/academic institutions. Consequently, availability of high-speed networks and fast general-purpose computers are facilitating towards the mainstream adoption of parallel computing.

However, it must be noted that development of parallel programs is complex. This has always been one of the major obstacles to the mainstream adoption of parallel computing. There are diverse reasons for the aforementioned difficulty in parallel programming:

- There is no single standard architecture and standard programming model for parallel computing. Unlike sequential computers, which follow the Von Neumann model of computation, different parallel architectures support different parallel programming models (e.g., data-parallel, data-flow, control-parallel, systolic). Each programming model gives birth to a group of new languages, compilers, compilation techniques and a group of programmers proficient in their use.

- There is no standard to measure the performance of a parallel program, in theory or in practice. Sometimes, researchers use *LogP* [2] or its descendant models but they are not always sufficient to model all possible parallel programming paradigms or models.

- There exists no standard parallel programming tool to develop, debug and profile parallel programs. As a result, the parallel programmers often develop their own specialized tools.

- For long time, researchers has been exploring different programming languages for parallel programming. Many extension to existing programming languages (*C-Linda* [3], *HPF* [4], *CC++* [5], etc) as well as many freshly designed programming languages (*NESL* [6], *Orca* [7], *PARLOG* [8], *ZPL* [9], etc) has been proposed. As a consequence, it requires higher learning time for the programmers to learn the language extension or the new language. At the same time, the codes written in those languages are not always reusable.

Numerous researches have been conducted to overcome some of the above mentioned difficulties. This research focuses on a specific approach to network-oriented parallel (cluster) programming that is based on frequently used parallel design patterns.

1.1 Pattern-based Parallel Programming

The concept of design patterns has been used in diverse domains of engineering, ranging from architectural designs in civil engineering [10] to the design of object oriented software [11, 12]. In the area of parallel computing, (parallel) design patterns specify recurring parallel computational problems with similar structural and behavioral components, and their solution strategies. Examples of such recurring parallel patterns are: static and dynamic replication, divide and conquer, data parallel computation with various topologies, compositional framework for control- and data-parallel computation, pipeline, singleton pattern for single-process (single- or multi-threaded) computation, systolic and wavefront computation.

Patterns in parallel computing have often been employed not only at the design level but also at the implementation level as reusable components. Several parallel programming systems have been built with the intent to facilitate rapid development of parallel applications through the use of design patterns. Some of these systems are *CODE* [13], *Frameworks* [14], *Enterprise* [15], *Tracs* [16], *DPnDP* [17], *COPS* [18], *PAS* [19], and *ASSIST* [20].

Most of the previous researches in this direction focused on the algorithmic / behavioral aspects of patterns, popularly known as *algorithmic skeletons* [21]. Algorithmic skeletons are best expressed using the various functional and logic programming languages [22, 23].

In contrast, Parallel Architectural Skeletons (PAS) [19, 24] specifies the architectural/structural aspects of patterns. The PAS model describes the various attributes of a message-passing parallel pattern in a generic and application-independent fashion. Unlike algorithmic skeletons, architectural skeletons in PAS can be very well expressed using object-oriented language(s). For instance, the PAS model has been fully implemented using C++ without necessitating any language extension. Consequently, the PAS approach is well suited to the main-stream of parallel application developers where the popular object-oriented languages like C++ and Java are the languages of choice.

Application development using PAS is hierarchical, and is similar to conventional parallel programming using MPI [25] and PVM [26]. However, PAS provides the added benefit of re-usability. A developer, depending upon the specific needs of a parallel application, chooses the appropriate skeletons, supplies the required parameters and application-specific code. Architectural skeletons supply most of the code that is necessary for the low-level and parallelism-related issues. In other words, architectural skeletons take care of application-independent parallel programming aspects whereas the developer largely supplies the necessary application code. Consequently, there exists a clear separation between application dependent code and application independent issues (i.e., *separation of concerns*).

1.2 Objective of This Research

Though reusability is a useful benefit, the lack of extensibility and the lack of support for pattern composition are some of the major concerns associated with most of the pattern-based approaches to parallel programming, including PAS. Most existing systems support a limited and fixed set of patterns that are hand-coded into those systems. Generally, there is no provision for adding a newly designed skeleton without understanding the entire system and writing the skeleton from scratch (i.e., lack of extensibility). So, if a required parallel computing pattern demanded by an application is not supported, generally the designer has no alternate way but to abandon the idea of using the particular approach altogether (lack of flexibility).

SuperPAS, introduced in [27], is an extension of the PAS system and it addresses the drawbacks mentioned previously. It provides a model for designing new skeletons. Besides, it also supports a skeleton description language (SDL) to realize the SuperPAS model for designing skeletons. Using SuperPAS, a skeleton designer can extend PAS by adding new skeletons to the repository (i.e., extensibility). SuperPAS also makes the PAS system more flexible by defining composition of skeletons, where two or more existing skeletons can be composed into a new skeleton.

The SuperPAS model and environment are targeted for two different groups of users: (1) *skeleton designers* who design new skeletons from scratch or through composition, and add the new skeletons to the skeleton repository, (2) *Application developers* who use the skeletons already available in the skeleton repository. Unlike a skeleton designer, an application developer does not need to have any knowledge about the SDL and he/she can directly develop application using C++. On occasions, a skeleton developer and an application developer may be the same person.

1.3 Organization of the Thesis

The thesis describes the SuperPAS model, the associated SDL and its use. The thesis is organized as follows: Chapter 2 provides a brief introduction to the PAS model. Chapter 3 introduces the SuperPAS model and necessary theories. The following chapter describes the SDL along with its grammar. That chapter also includes design of several example abstract skeletons and development of one application in SuperPAS system. In chapter 5, we discuss the performance issues of SuperPAS-based applications from different perspectives. Finally chapter 6 concludes the thesis with a discussion about future directions.

Chapter 2

Preliminaries

This chapter is divided into two sections. In the first section, we briefly introduce several pattern-based parallel programming systems. The second section is dedicated to an introduction of the PAS [19, 24] system. That section also discusses different components of the PAS system that are going to be used in rest of the thesis.

2.1 Pattern-Based Parallel Programming Systems

In this section, we give a very brief introduction to several pattern-based parallel programming systems. In the following subsections, we discuss about Frameworks [14, 28], Enterprise [15], COPS [18] and *ASSIST* [20].

2.1.1 Frameworks

In the late 80s Frameworks [14,28] was developed at the University of Alberta. Frameworks is an early system that successfully exploited the idea of using commonly occurring parallel structures in parallel application development. It was specifically designed to restructure existing sequential programs to exploit parallelism on workstation clusters. The Frameworks programming model supports separation of specifications by segregating the application specific sequential code from the parallel structure of the application, which can be developed separately.

Patterns in Frameworks are called templates. In the Frameworks programming model, an application consists of a set of modules that interact with one another via calls similar to remote procedure calls (RPCs). Messages between modules are in the form of user defined frames. Each module consists of a set of procedures, one of which is the entry procedure and is the only procedure called by other modules in the application. A module also contains local procedures, callable only within the module. Developers create modules by selecting appropriate templates and application procedures. Arbitrary process graphs can be created by interconnecting resulting modules. Each module is written in an extension of C, augmented by features to support remote procedure calls.

2.1.2 Enterprise

Enterprise [15] was also developed as a successor to Frameworks. It is not just a parallel programming tool; it is a complete parallel programming environment with a complete tool set for parallel program designing, coding, compiling, executing, debugging and profiling.

There are a number of improvements in Enterprise over Frameworks. Patterns in Enterprise are at a much higher level of abstraction than in Frameworks. The three-part templates in Frameworks are combined into single units in Enterprise and are called *assets*, which are named to resemble operations in a human organization. For example, the asset named *department* represents a master-slave pattern in the traditional parallel programming terminology. A fixed collection of assets is provided by the system, which can be combined to create an asset diagram to represent the parallel program structure. Each asset is associated with a piece of application code consisting of procedures with sequential flow of control. Assets can be hierarchically combined to form a parallel program. Features like *future* variables enable more concurrency. A number of other features and tools improve the usability and portability aspects of the system.

2.1.3 COPS

Again, COPS [18] system was developed at the University of Alberta. COPS is a layered development system, where the uppermost layer provides the parallel design pattern abstraction to the developer. At this layer, a user subclasses a pattern and fills in the hook methods to reflect application specific details. The two lower layers are for performance tuning, where the developer can look at the generated code (both in an intermediate language and a native language) and fine tune to the needs of the final program. Unlike Frameworks and Enterprise, COPS is a parallel programming model for shared memory systems.

2.1.4 ASSIST

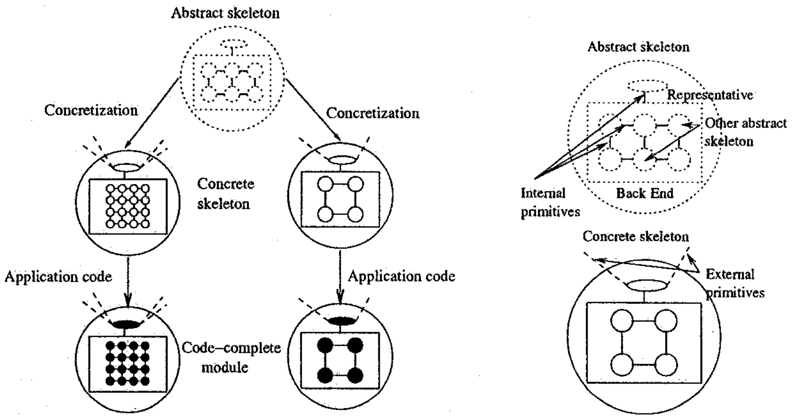
Lastly, *ASSIST* [20], developed at University of Pisa, is a skeleton-based system that allows a user to define arbitrary graphs, whose nodes are either sequential or parallel modules. Those parallel modules are denoted as *parmods*. The edges of the graph are channels of interactions among parmods. The edges have the semantics of channel of stream. Parmods are defined to be able to express the semantics of more common data-parallel and task-parallel patterns. Development in ASSIST is bottom-up where parmods are flat graph connecting virtual processors. ASSIST provides a way to design reusable components. A reusable component is a graph and when reused, the user must correctly interface that component with other components to have a complex program.

2.2 The PAS System

Goswami et. al. introduced Parallel Architectural Skeletons or PAS [19, 24] at the University of Waterloo. This system specifies the architectural/structural aspects of patterns as skeletons. A *skeleton* in PAS encapsulates the structural/architectural attributes of a set of specific patterns in parallel computing. Each PAS skeleton is parameterized, where the value of a parameter is determined during the application development phase. As an example, a k-dimensional data-parallel Mesh skeleton, provided by PAS, encapsulates the structural aspects of a data-parallel Mesh pattern, together with the associated communication-synchronization primitives. Some of the parameters of the skeleton are: the number of dimensions of the mesh (i.e., k), and the length of each dimension. These parameters are bound to actual values during the application development phase.

During the rest of the discussion, a PAS skeleton with unbound parameters is called an *abstract skeleton* or an *abstract module* (the term *module* reflects the modular structure of a skeleton, and will be discussed shortly). An abstract skeleton becomes a *concrete skeleton* or a *concrete module*, when the parameters of the skeleton are bounded to actual values during the application development phase. A concrete skeleton is yet to be filled with application-specific code. A concrete skeleton which is completely filled with application-specific code is called a *code-complete parallel module* or simply a *module* (the term *skeleton* is omitted here because with application code, it is no longer a skeleton). As it will be discussed shortly, a parallel application is a hierarchical collection of modules.

Figure 1(a) roughly illustrates the various phases of application development using PAS. As shown in the figure, different parameter bindings to the same abstract skeleton can result in different concrete skeletons. A concrete skeleton inherits all the properties associated with the abstract skeleton. Besides, it has proper values bound to each of the parameters, depending on the needs of a given application. In object-oriented terminologies, an abstract skeleton can be described as the *generalization* of particular design patterns. A concrete skeleton is an application-specific *specialization* of a skeleton.



(a) Abstract skeleton, concrete skeleton and code complete module

(b) Different components of a skeleton

Figure 1: PAS skeletons and their components

Irrespective of the pattern type, an abstract module (i.e., an abstract skeleton), A_m, consists of the following set of attributes. Figure 1(b) diagrammatically illustrates the attributes of an abstract skeleton and a concrete skeleton, where the skeleton represents a 2-D Mesh topology.

- *Representative* represents the module in its action and interactions with other modules. Initially, the representative is empty and is subsequently filled with application-specific code (refer to the following discussion).

- *Back-end* of an abstract module A_m can formally be represented as $\{A_{m1}, A_{m2}, \ldots, A_{mn}\}$, where each A_{mi} is itself an abstract module. The type of each A_{mi} is determined after the abstract module A_m is concretized. Note that collection of concrete modules inside another concrete module results in a (tree-structured) hierarchy. Consequently, each A_{mi} is called a *child* of A_m, and A_m is called the *parent*. The children of an module are *peers* of one another. In this literature, the children of a skeleton are also referred as *computational nodes* of the skeleton or associated patterns.

- *Topology* is the logical connectivity between the modules inside the back-end. It also includes the connectivity between the children and the representative.

- *Internal primitives* are the pattern-specific communication / synchronization / structural primitives. Interaction among the various modules is performed using these primitives. The internal primitives are the inherent properties of the skeleton and they capture the parallel computing model of the patterns as well as the topologies.

There are pattern-specific parameters associated with some of the previous attributes. For instance, if the topology is a Mesh, then the number of dimensions of the Mesh is one parameter, and the connectivities for the nodes at the edges (i.e., wrapped around / unwrapped) are another parameter. Fixing these parameters, based on the needs of an application, results in a concrete module. A concrete module C_m becomes a code-complete module when (i) the representative of C_m is filled in with application-specific code, and (ii) each child of C_m is code-complete. This description obviously indicates that application development using PAS is hierarchical.

Clearly, all of the previously described attributes of an abstract skeleton are inherited by the corresponding concrete skeletons as well as a code-complete modules. In addition, we define the term *external primitives* of a concrete or a code complete module as the set of communication / synchronization / structural primitives using which the module (i.e., its representative) can interact with its parent (i.e., representative of the parent) and peers (i.e., representatives of the peers). Unlike internal primitives, which are inherent properties of a skeleton, external primitives are adaptable, i.e., a module adapts to the context of its parent by using the internal primitives of its parent as its external primitives. While filling in the representative of a concrete module with application-specific code, the application developer uses the internal and external primitives for interactions with other modules in the hierarchy. Examples of some of these primitives for a Mesh structured topology are *SendToNeighbor(...)*, *RecvFromNeighbor(...)*, *ScatterPartitions(...)*, *GatherResults(...)*, etc. Chapter 4 includes descriptions of several example skeletons along with their different attributes.

A parallel application developed using PAS is a hierarchical collection of (code-complete) modules. The root of the hierarchy, i.e., a code-complete module with no parent, represents a *complete parallel application*. Each non-root node of the hierarchy represents a partial parallel application. Each leaf of the hierarchy is called a *singleton module* (and correspondingly, a *singleton skeleton* for the abstract counterpart).

Interactions among modules are based on pattern-specific message-passing primitives, which make the PAS model suitable for a network cluster. The high-level abstractions provided by a skeleton hide most of the low-level details which are commonly encountered in any parallel application development.

In this chapter, we briefly described several pattern-based parallel programming systems. We also discussed about the PAS model, on top of which SuperPAS model is designed. Interested readers can find a comprehensive description of the PAS model, detailed examples and comparison with other pattern-based systems in [19, 24]. In the next chapter, we give an informal introduction to the SuperPAS system.

Chapter 3

Introduction to SuperPAS

In this chapter, we introduce SuperPAS. We start in the next section by describing the reasons that motivated us to research towards SuperPAS. Section 3.2 categorizes the users of the SuperPAS system and distinguishes their roles. Section 3.3 describes the steps involved in the development of a parallel application using SuperPAS. Section 3.4 discusses the model of the SuperPAS system. Finally, a comparison between SuperPAS and the similar systems is given in Section 3.5.

3.1 Motivation

Like most other pattern-based parallel programming systems, the original PAS system repository of (abstract) skeletons was built by hand-coding and there was no provision for adding new skeletons without writing them from scratch using the associated high-level programming language (e.g., C++). The drawback of this approach is that writing a skeleton from scratch is not easy. It requires in-depth knowledge of the programming model and the implementation of the entire system. This is the reason that the original PAS and all other similar systems are not extensible or very difficult to extend.

One of the motivations behind SuperPAS is to make PAS extensible. In a nutshell, SuperPAS is an extension of the PAS model that facilitates design of abstract skeletons from scratch. It includes a *Skeleton Description Language* (SDL) to describe an abstract skeleton according to the SuperPAS model. Using the SDL, a skeleton designer can add new abstract skeletons to the skeleton repository with minimum efforts. Using it, the designer can also specify skeleton specific primitives and parameters. Moreover, the SuperPAS model allows a skeleton designer to compose two or more existing skeletons into a new composite skeleton. More about composite skeleton is discussed in Section 3.4.

3.2 Categorization of the Users of SuperPAS System

SuperPAS distinguishes the users of the system into two groups. The *skeleton designers* (in short, *designers*) are in the first group. A designer designs and implements required skeletons using SuperPAS SDL. A designer is supposed to be a parallel programmer, who has good understanding in writing parallel programs as well as has in depth knowledge of the SuperPAS model and the SDL.

The second group of users is the *application developers* (in short, *developers*). Application developers use the skeletons, designed by the skeleton designers, to develop various parallel applications. Due to the availability of the required skeletons in the skeleton repository, the application development procedure becomes simple. Application developers are also parallel programmers with very little or no knowledge about the SuperPAS SDL.

The degree of experience and expertise of a skeleton designer is assumed to be higher than an application developer. In fact, in an organization, one person can be both skeleton designer and application developer at the same time. From this categorization, it is clear that the original PAS was targeted to the application developer group. It provided a systematic and natural way of using parallel skeletons to develop parallel applications. On top of the PAS system, SuperPAS targets the users in the skeleton designer group and provides them a systematic way to design abstract skeletons.

The SuperPAS system requires that the designers not only investigate the properties of a skeleton and design it, but also ensure the correctness of the designed skeleton. This is an important requirement to ensure that the application developers concentrate only on the development phase rather than worrying about violating constraints of the used skeletons, while developing the parallel application.

3.3 Development Steps in SuperPAS

In SuperPAS, the development process starts with a skeleton designer writing the abstract skeletons using the SDL. During the design step, the designer determines the structures, parameters and primitives of the patterns to be implemented in the skeleton. Then the designer expresses those properties of the patterns of the skeleton in SDL and this results in an abstract skeleton. The designer may then store the designed skeletons in the skeleton repository. As is discussed in the previous section, it is the responsibility of the designer to ensure the correctness of the designed skeleton. Correctness of a skeleton is ensured by confirming the correctness of the topology of the skeleton as well as the correctness of the primitives for any allowable values of the parameters of the skeleton.

When an application developer designs and develops a parallel application, she chooses the proper skeletons from the skeleton repository, concretizes them, and finally fills them with application specific code to create the final parallel application. To concretize an abstract skeleton written in SuperPAS SDL, the developer can either directly modify the abstract skeleton code (in SDL) or use the provided tools. Then the developer uses the SuperPAS tools to generate C++ codes for the concretized skeletons. In fact, there is no semantical differences between the generated C++

concrete skeletons and concrete skeletons of the original PAS system.

To help the designer and the developer of the system, the SuperPAS programming environment provides standard tools to: (1) verify the SDL syntax of an abstract skeleton, (2) concretize an abstract skeleton, and (3) translate a concrete SDL skeleton into C++ code. The generated C++ code framework for concrete skeletons provides a complete object-oriented interface for the application developers. The SuperPAS tools and the SuperPAS runtime system hide most of the underlying complexities of the system.

3.4 Introduction to the SuperPAS Model

In this section, we introduce the SuperPAS model in an informal way. We describe the model with several short examples for ease of understanding.

3.4.1 Overview

SuperPAS model incorporates PAS model and adds extra layers over it. This introduces a new type of users to the PAS system (skeleton designer), whose sole job is to design new abstract skeletons. The views of a skeleton designer and a application developer can be best illustrated with the views of a microprocessor designer and a microprocessor user (programmer). While designing a microprocessor, a designer perceives the SSI and MSI circuits as the basic building blocks (primitives). Whereas a user of that processor can perceive only the instructions designed into the processor, not the SSI / MSI circuits.

In SuperPAS, a skeleton designer is provided with a set of abstract computational nodes and a rich set of basic communication and synchronization primitives to establish communication and synchronization among the nodes. At first, the designer decides about the parallel patterns to be implemented into each intended new skeleton. Then she maps the parallel components of each pattern onto the available computation nodes. She also establishes connectivities among the mapped nodes to reflect the connectivities among the parallel components of the pattern. Those connectivities are built on top of the basic connectivity (communication / synchronization) primitives provided by the SuperPAS system. Those higher level primitives are specific to the associated pattern (and hence skeleton). Mapping the structures of all the patterns of the skeleton onto the abstract computational nodes, along with the higher level pattern specific primitives, results in an abstract skeleton.

Now, when an application developer picks up an abstract skeleton, she can perceive only the abstract topology and pattern specific primitives of each of the patterns of the abstract skeleton. The fine grain building blocks (basic abstract computational nodes and basic communication primitives) are completely hidden from her. Rest of the section elaborates this overview in details.

3.4.2 The Virtual Processor Grid

SuperPAS provides a set of multidimensional grids (described as abstract computational nodes in the last subsection) to embed the topology or structure of patterns of an abstract skeleton. Usually one pattern is embedded onto one multidimensional grid. Each node of the grid can be considered as a virtual processor. In this thesis, we use the terms *node of a grid* and *virtual processor* interchangeably. The multidimensional *virtual processor grids* or *VPGs* are equipped with generic communication primitives. Those primitives include primitives for synchronous and asynchronous peer-to-peer communications. Moreover, collective or group communication and synchronization primitives like *broadcast, scatter, gather, reduce, all-to-all send-receive* and *barrier* are also supported.

We make a very careful decision about choosing the set of basic communication primitives for the VPGs. We choose to make our primitives a super set of the communication primitives in the prominent parallel programming environments. Our choice is influenced by the research article [29], MPI standard [25], PVM documentations [26] and our experience with PAS and other pattern-based systems.

Furthermore, the choice of the regular grid structure for the virtual processor grid (VPG) is also not arbitrary. There are several factors which influenced the selection of this structure:

1. A grid is a simple regular structure, which enables a uniform addressing schema to address each node of the grid.

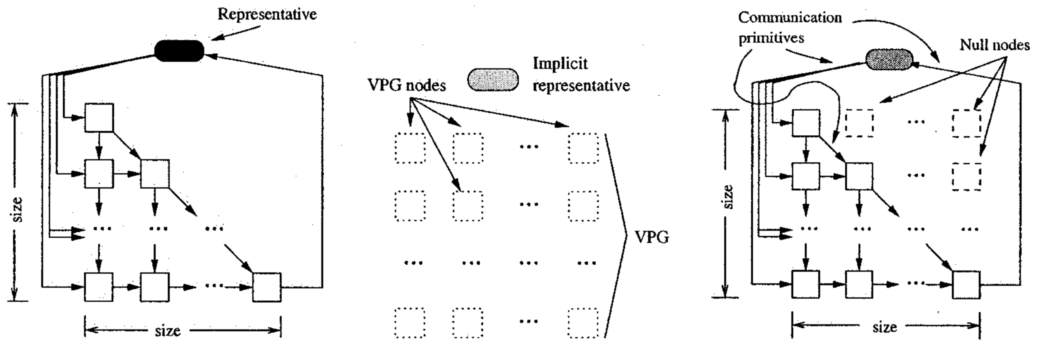
2. The processor grid is a very popular parallel processor structure. A wide collection of suitable communication-synchronization primitives for such structure can easily be found in the existing literature.

3. A regular structure, commonly associated with a pattern, can be easily unfolded and mapped onto another regular structure (i.e., onto a grid).

In the existing literature [30–32], many of such mapping of pattern onto processor grid can be found. Moreover, an irregular structured pattern (e.g., an arbitrary graph) can also be mapped onto a grid by explicitly mapping each structural block, e.g., each node and associated connectivities of that pattern onto the virtual processor grid explicitly.

3.4.3 Mapping A Pattern onto A VPG

Now, when the designer wants to design a skeleton, at first she needs to map the structure of each of the patterns of the skeleton onto a VPG. Figure 2 shows such a mapping. In this example, the designer wants to design an abstract *Wavefront* skeleton which is actually an implementation of the *Wavefront* pattern. Figure 2(a) is the visualization of the Wavefront pattern. Note that, a visualization of a pattern gives the abstract view of the pattern. At the same time, it shows the abstract topology of the parallel components of the pattern.



(a) Wavefront Pattern ·(b) The VPG (c) Wavefront Skeleton

Figure 2: Mapping Wavefront Pattern in a VPG

From a visualization, the designer has to make several design decisions. At first, she should decide about the *parameters* of the pattern. For example, the structure of a Wavefront pattern becomes generic if the size (the number of rows or the number of columns) of the pattern is considered to be a parameter rather than some fixed constant. In this example, we designate this parameter as *size*.

In Wavefront, the choice of a two dimensional VPG (refer to Figure 2(b)) is (probably) obvious because it provides an one-to-one mapping of the computational nodes of the pattern onto the nodes of the VPG. Finally, Figure 2(c) shows the actual mapping. From the figure, it can be found that even after limiting the height and width of the VPG (to the parameter *size*), there are virtual processors where no computational node of the pattern is mapped onto. Those virtual processors are called *null virtual processors* or *null nodes*.

The embedding of a pattern onto a VPG is complete when the associated communication, synchronization and structural primitives are defined. In Figure 2(c), some of the communication primitives are marked. Examples of communication primitives in a Wavefront pattern (as well as in as Wavefront skeleton) are: (1) receive a message from the representative, (2) send a message to the left neighbor, (3) receive a

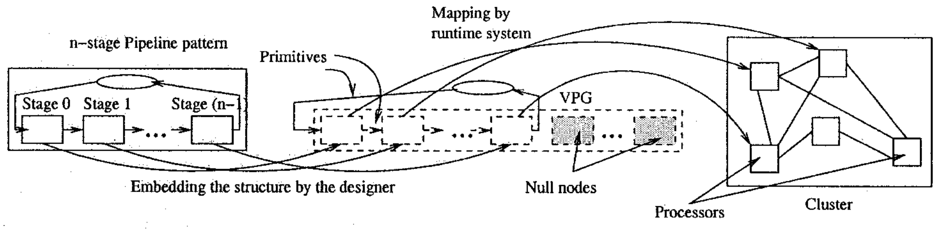


Figure 3: Mapping of $\mathcal{AT}$ of a Pipeline pattern onto a $\mathcal{VPG}$ and the $\mathcal{VPG}$ onto physical processors

message from the top neighbor, etc. Examples of structural primitives are: (1) is a node located at the first column, (2) is a node located at the last column, (3) is a node located on the diagonal, etc.

3.4.4 Denotations

In this subsection, we formally introduce some terminologies that we are going to use throughout the thesis. In the SuperPAS model, the (abstract) structure of a pattern (as in Figure 2(a)) is designated as *abstract topological space* ($\mathcal{AT}$) of the pattern. The abstract topological space is composed of zero or more abstract parallel computational nodes of the pattern along with their connectivities (represented by a connectivity function $\mathcal{T}$). The mapping function, $\mathcal{M}$, maps nodes of $\mathcal{AT}$ onto the virtual processors of the *VPG space* (designated as $\mathcal{VPG}$). Provided $\mathcal{M}$ and $\mathcal{T}$, it is easy to establish the connectivities among the mapped non-null virtual processors of the $\mathcal{VPG}$ and can readily be expressed as $\mathcal{M}.\mathcal{T}.\mathcal{M}^{-1}$. The embedding of a $\mathcal{AT}$ onto a $\mathcal{VPG}$ results in a *abstract mapped space* (designated as $\mathcal{P}$), as shown in Figure 2(c). Note that, $\mathcal{P}$ contains only the non-null nodes and hence represents the back-end of a skeleton. This is to mention here that null-nodes do not impose any extra overload on the run-time system, as no resources are allocated for them. The approach taken by SuperPAS is shown pictorially in Figure 3 in the process of developing a parallel application involving a *Pipeline* skeleton.

3.4.5 Primitives

In SuperPAS, the connectivity function ($\mathcal{T}$) and the mapping function ($\mathcal{M}$) are used together to design pattern specific primitives. Combining all primitives from all the patterns, implemented in a skeleton, results in skeleton specific internal primitives. SuperPAS divides the primitives of a pattern (and in-turn of an abstract skeleton) into two categories: *private* and *public* primitives. The private primitives of a pattern can only be used by the representative of the associated skeleton and are not inherited by the children of that skeleton. On the other hand, public primitives are available only to the children of the skeleton as external primitives. In case of the Wavefront pattern, *receiving a message by the representative from the node at last column* is a private primitive whereas *sending a message to the left neighbor* is a public primitive.

3.4.6 Composition

SuperPAS model supports the idea of composition of abstract skeletons. Composition is the way to compose simpler abstract skeletons into a complex one. In the following paragraphs, we describe the motivation behind incorporating the idea of composition as well as the model of composition.

Motivation Behind Composition

A large-scale parallel application is often a composition of multiple patterns. Sometimes it is more desirable to have a single composite skeleton rather than a collection of smaller skeletons, provided that the composite skeleton will be used for developing variations of similar applications, i.e. the composite skeleton will be a reusable component.

A typical example can be found in the domain of image processing applications. In many cases, such an image processing application can be divided into following three stages:

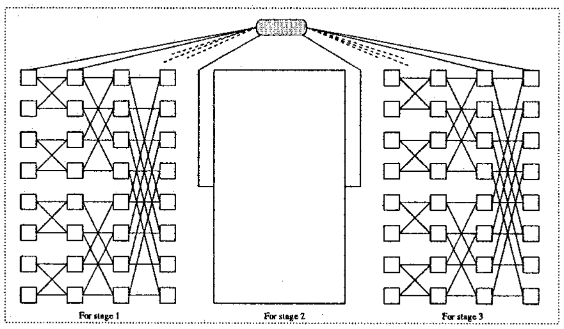


Figure 4: A composed skeleton for image processing

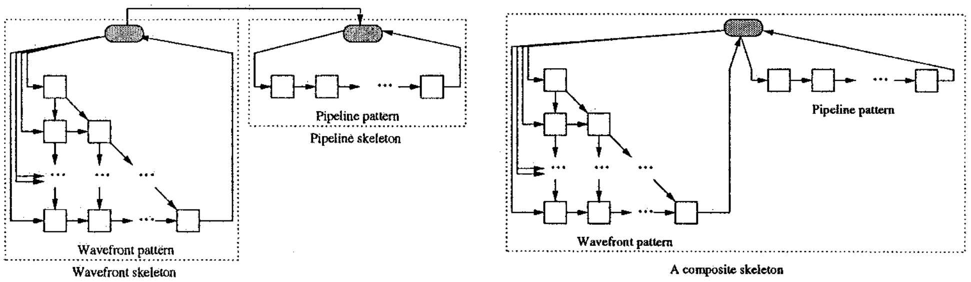
1. Convert the image from pixel domain to frequency domain using (Forward) Fourier Transformation

2. Do some specific operations (examples: convolution, de-noising of the transformed image, etc) and

3. Finally, get the processed image in pixel domain using Inverse Fourier Transformation

A parallel (Forward and Inverse) Fourier Transformation algorithm can be implemented using the *Cube Connected Cycles* (CCC) [33] or the *Butterfly* patterns. Operations like convolution and de-noising are usually implemented using data-parallel, master-slave or some other simple and / or complex patterns. The three steps of image processing can be composed into a single skeleton as is shown pictorially in Figure 4.

Readers should note that composition is different from construction of skeleton hierarchy during concretization. Composition is performed on abstract skeletons to create a composite abstract skeleton. Composition is performed by the skeleton designers whereas the skeleton hierarchy is constructed by the application developers. Composition may require an in-depth knowledge of the *abstract mapped spaces*, whereas the creation of the skeleton hierarchy does not. Mapping a concrete skeleton onto a child



(a) Two separate skeletons

(b) A composite skeleton

Figure 5: Composing skeletons towards performance

of another concrete skeleton usually increases the height of the skeleton hierarchy in an application. However, use of composite skeletons to develop applications usually reduces the height of the skeleton hierarchy.

Another reason for having composite skeleton is performance. Let us consider the example in Figure 5(a), where a Wavefront and a Pipeline skeleton are shown. The output of the rightmost child of the Wavefront is sent back to the representative; the representative routes it to the representative of the Pipeline skeleton, which in-turn again routes to the first stage of the Pipeline. Composition of these two skeletons is shown in Figure 5(b). From the figure, it is evident that in this example, composition reduces the number of routing requirement by 1 as compared to the skeletons in Figure 5(a).

Model of Composition

In simple word, composition of skeletons S_i and S_j into skeleton S_k results in a union of the parameters and abstract mapped spaces in S_i and S_j. Say, skeleton S_i is composed of abstract mapped spaces $\mathcal{P}_{1i}$, $\mathcal{P}_{2i}$, $\ldots$, $\mathcal{P}_{mi}$ and S_j is composed of $\mathcal{P}_{1j}$, $\mathcal{P}_{2j}$, $\ldots$, $\mathcal{P}_{nj}$. Then S_k will be composed of $\mathcal{P}_{1i}$, $\mathcal{P}_{2i}$, $\ldots$, $\mathcal{P}_{mi}$, $\mathcal{P}_{1j}$, $\mathcal{P}_{2j}$, $\ldots$, $\mathcal{P}_{nj}$.

We define the *skeleton space* ($\mathcal{S}$) of a skeleton as a space which is exactly big

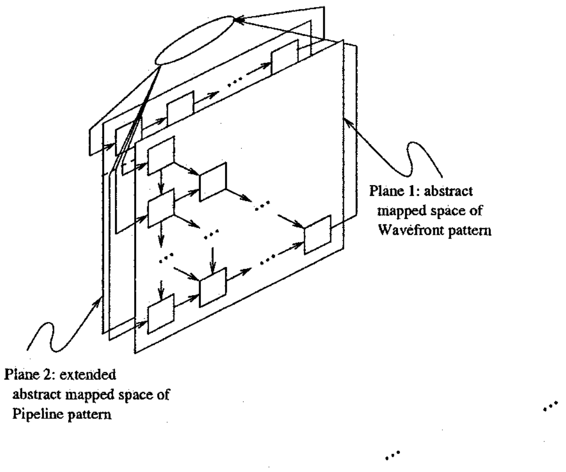

Plane 1: abstract mapped space of Wavefront pattern
Plane 2: extended abstract mapped space of Pipeline pattern

Figure 6: The skeleton space

enough to hold all the abstract mapped spaces of that skeleton. Formally, let us assume that a skeleton S consists of the abstract mapped spaces $\mathcal{P}_1$, $\mathcal{P}_2$, $\ldots$, $\mathcal{P}_N$. Assuming that the abstract mapped space $\mathcal{P}_i$ is a k_i dimensional space (i.e., result of mapping an abstract topological space of a patten onto a k_i dimensional $\mathcal{VPG}$), the skeleton space, $\mathcal{S}$, would be of a $K = \max\{k_i \mid 1 \leq i \leq N\} + 1$ dimensional space. Semantically, to create the skeleton space, an abstract mapped space $\mathcal{P}_i$ is extended from k_i dimension to $K-1$ dimension. While extending the dimension, the higher $K - 1 - k_i$ dimensions are made limited to size 1 to ensure consistency. The length of the K-th dimension of the skeleton space, $\mathcal{S}$, is N and the extended abstract mapped space of $\mathcal{P}_i$ is placed on the i-th entry of the K-th dimension. Note that, the application developer can perceive only the skeleton space and the associated abstract mapped spaces. Further detail remains hidden from her.

Figure 5(b) is redrawn in Figure 6 to reflect the idea of the skeleton space. The skeleton space includes the abstract mapped spaces of the Wavefront pattern and the Pipeline pattern. Among those abstract mapped spaces, the mapped space for the Wavefront is of the highest dimension, which is two. As a result the skeleton space is of three dimension. As shown in the figure, the first plane of the skeleton

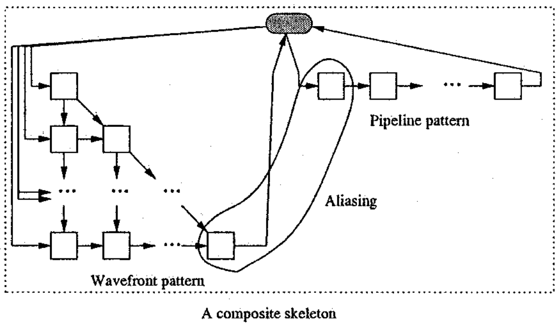


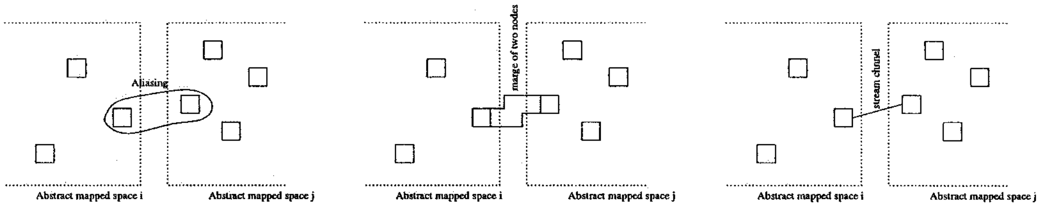
Figure 7: Aliasing in a skeleton

space includes the mapped space of the Wavefront pattern whereas the second plane includes the mapped space of the Pipeline pattern.

In order to achieve more flexibility, SuperPAS provides *aliasing.* Aliasing is the way to combine two nodes from two different abstract mapped spaces of a particular skeleton. The idea is shown in Figure 7. Aliases in SuperPAS are expressed using *aliasing function* (designated as $\mathcal{A}$). The choice of a function rather than a generic relation is governed by simplicity of understanding and use.

SuperPAS provides two modes of aliasing: (1) *fusion* paradigm and (2) *linkage* paradigm. In the linkage paradigm two aliased nodes are connected via a channel and both of the nodes remain as separate entities. On the other hand, in fusion paradigm, two aliased nodes are unified into one node. As a result, that unified node becomes members of both of the abstract mapped spaces, where the original nodes belong to. Examples of fusion and linkage paradigm of aliasing are shown in Figure 8.

To describe the idea formally, let us assume that $\mathcal{S}_S$ is the skeleton space of skeleton S and $\mathcal{P}_i$, $\mathcal{P}_j$ and $\mathcal{P}_k$ are three abstract mapped spaces of S. Lets $P_l \subseteq \mathcal{P}_l$ where $l \in \{i, j, k\}$. The aliasing function is defined as, $\mathcal{A} : P_m \rightarrow P_n$ where $m, n \in \{i, j, k\} \wedge m \neq n$. Say, $\mathcal{A}(p_i) = p_j$ and $\mathcal{A}(p_j) = p_k$, where $p_l \in P_l \wedge l \in \{i, j, k\}$. In the fusion paradigm of model, those two aliasing imply $\mathcal{A}(p_i) = p_k$. However, this implication is not true for the linkage paradigm.



(a) Two abstract mapped spaces

(b) Aliasing in Fusion paradigm

(c) Aliasing in Linkage paradigm

Figure 8: Fusion and linkage paradigm of aliasing

3.4.7 Labeling

In PAS, a labeling function, $\mathcal{L}$, labels each of the nodes (children in the back-end) of a concrete skeleton, CS, with instances of other abstract skeletons. Labeling can also be considered as specifying the types of the children in the back-end. Now, say, in an occasion $\mathcal{A}(p_i) = p_j$ and $\mathcal{L}(p_i) = AS_s$, where AS_s is an instance of an abstract skeleton. If the aliasing function, $\mathcal{A}$, follows the fusion paradigm, $\mathcal{L}(p_j)$ must also be AS_s. However, in linkage paradigm of aliasing, p_j can be labeled without considering the labeling of p_i, as in this paradigm p_i and p_j are considered to be two separate entities.

3.5 Comparison with Related Works

Most of the pattern-based systems proposed in the literature do not address the issue of extensibility. This issue has been addressed by a few recent systems only, such as: *ASSIST* [20] and *MetaCOPS* [34]. In this section, we compare SuperPAS with both of the systems. At first, we revisit the ASSIST and MetaCOPS programming models. Then all the three systems are analyzed from the viewpoint of a parallel application developer.

3.5.1 MetaCOPS

COPS is one of the most recent pattern-based parallel programming environments. MetaCOPS is the tool that made COPS system extensible. MetaCOPS takes a generative design pattern approach to design a new pattern for the COPS system. The main target of the COPS as well as the MetaCOPS system are the ability to incorporate user-defined capabilities (generality), the ability to adapt all possible architectures and above all, the ability to generate applications that can be tuned for high performance computing.

COPS divides the design process of a parallel application into several steps. At first, the designer identifies the design pattern required for the target parallel application and picks the pattern template from the library. Then she adapts the selected pattern template by choosing proper values for each of the parameters of the template. A template parameter can take one of values specified by the designer. This application oriented customization makes possible to generate optimized framework code through a conditional code generation procedure.

The application developer then provide the application-specific sequential hook methods and other non-parallel code to build the final parallel application. Then performance of the parallel application is monitored and if it is not satisfactory, generated parallel code is inspected at a lower layer of abstraction where all the a high-level and explicit parallel object-oriented code is exposed. The developer needs to identify and modify the code that is clogging the performance. At the last step, the performance of the application is re-monitored and if it is still not acceptable, the application developer needs to modify the implementation of the used high-level primitives optimized for her target architecture.

At this point, it is essential to point out that template parameters in COPS are not same as the skeleton parameters in PAS. A template parameter can take one of the values, specified by the template designer. Moreover, parameters in COPS provides choices from the behavioral perspective. For example, in a mesh pattern template, each mesh element can have either four or eight neighbor and depending on the choice, made by an application developer, each node communicates with either four or eight neighbors.

MetaCOPS provides a GUI-based interface to design a pattern template. Using MetaCOPS, a designer specifies a placeholder name of the template to design. She also specifies the parameters of the template and all allowable values for each of those parameters. The designer then needs to write code for all possible combination of choices of parameters. In fact, the COPS system does a conditional code generation based on the codes written by the designer and values of the parameters chosen by the user. This approach ensures correct and optimized code for all possible instances of a particular pattern template.

The initial implementation of COPS as well as MetaCOPS was for shared memory multi-processor systems. The applications developed in COPS were multi-threaded Java applications. As the major parallel architectures are distributed memory systems (clusters of workstation) [1], authors of COPS extended it for those systems, keeping the original interfaces unchanged [35].

3.5.2 ASSIST

ASSIST is a programming environment targeted to the development of parallel and distributed high-performance applications. The main goals of ASSIST are: high-level programmability and software productivity for complex multidisciplinary applications, performance portability across different platforms and finally effective reuse of existing parallel software components.

In the ASSIST programming model a parallel program can be expressed by a generic graph, where each node or component can be a parallel module or a sequential module. The parallel module or *parmod* is able to express the semantics of more common data and task parallelism and nondeterminism, frequently found in the parallel patterns. Composition of parallel and sequential modules is performed through edges of the graph where an edge is represented by a stream, i.e. ordered sequences, possibly of unlimited length, of typed values. The ASSIST model allows each of the parallel components of an application to communicate with external components.

A composed modules, expressed by a graph P, can be, reused as a component of a more complex component or program Q. The composition is correct provided that P is correctly interfaced to the other modules of Q, i.e. the types of input and output streams must be compatible.

An ASSIST sequential module represents the simplest form of behavior of a deterministic data-flow. A parmod consists of a set of virtual processors which are independent or cooperating entities. A set of virtual processors act as parallel computing engines. Different virtual processors in a parmod can perform same or different tasks. If there is an input at the input-section of a parmod and if the input satisfies the input constraints, the parmod becomes active.

Implementations of the ASSIST model for network of workstations and Grids are reported [20,36]. The users of the system use a coordination language, called ASSIST-CL, to develop applications in ASSIST. Actions performed by a virtual processor or a sequential module is written in either C, C++ or in Fortran. ASSIST component can communicate with existing external CORBA components.

3.5.3 Comparison From a Developer's Perspective

In this subsection, we compare all the three systems from the perspective of a parallel application developer. Here, we assume that a parallel application is complex and is a composition of several patterns. While developing a parallel application, the user may want to replace a sequential component of the application with a parallel one or vice-versa due to the flexibilities or limitations of the underlying hardware architecture.

The COPS system suffers from the lacing support of pattern composition. It does neither supports composition during pattern design phase or application development phase. So, development of a complex parallel application requires to design a new pattern that represents the composition of all the patterns needed in the target application. In fact, it is hard to denote this newly designed pattern as *pattern* because it is too much application oriented (specialized). As a result, a developer needs to invest her time and effort to design the complex application from scratch even though the required building blocks of the application exist as several smaller patterns.

The ASSIST model is more relaxed from this perspective. It allows reuse of an already designed component, provided that they are interfaced properly. As a result, it is certainly possible to replace a sequential or parallel module with another sequential or parallel module without re-designing the whole application from scratch.

Unfortunately, ASSIST does not support nesting of parallel modules, i.e. a virtual processor of a parmod can not be replaced with another parmod. As a result, if a problem requires a parmod to be parallelized further, a re-design of the whole parmod is mandatory.

On the other hand, SuperPAS and PAS models allow composition of skeletons at both design and use phases. Either the designer can design a complex and big skeleton for a specific application or the developer can pick the required skeletons and build a skeleton hierarchy for the target application. The models provide inherent support for replacing a component of a program with another component, possibly composed of another hierarchy of components.

The ASSIST model supports only three kinds of topology among the virtual processors of a parmod, i.e. *multi-dimensional array*, *none* (they work independently of each other) and *one* (sequential component with features like non-determinism, etc.). The *multi-dimensional array* topology can easily express data parallelism whereas *none* topology can easily express independent task parallelism. But in real life, a parallel application is a complex composition of both of them. In ASSIST, it requires a lot of efforts to describe such complex compositional structures. In contrast, SuperPAS provides pattern independent communication / synchronization primitives and the designer expresses the required topology by designing proper high-level pattern specific primitives.

Neither of COPS and ASSIST supports design of patterns or reusable components with parametric structure (for example, a k dimensional mesh rather than a two dimensional mesh). Though COPS allows parametric behavior, ASSIST demands explicit declaration of connectivities among concurrent modules. As a result, the user of the patterns either needs to design her solution space conforming to the restricted structure of the given pattern in the repository or needs to design other patterns according to her needs. The SuperPAS model allows skeletons with parametric structures and topologies and the user just needs to tailor the required skeleton according to the application requirements by specifying proper values for the associated parameters.

The regular topologies (or structural patterns) found in the domain of parallel computing covers so great number of problems that researchers started to think about architectures that support those regularities. This resulted in architectures with specific processor topologies, for example processors connected in mesh or hypercube topology. But a computation of one pattern runs inefficiently on an architecture that follows a different pattern. Moreover, most of the parallel applications are compositions of more than one patterns. As a result, researchers found a way around to embed the patterns in software components. The SuperPAS model is partial in favoring the designers to describe such a regular structure (as discussed in Sub-section 3.4.2). without compromising the generality to express an arbitrary structure. As a pattern / skeleton design tools neither COPS or ASSIST favors the designer to design those regularities, found in the parallel patterns.

We believe that the intention of the ASSIST model is to develop parallel applications rather than reusable components. In the ASSIST model, an application or a parallel component is represented by a graph of sequential and / or parallel modules, where the topology among the modules are fixed. The model allows the actual computation, performed by each of the modules, to be left empty (to be fillable by the end user), at the same time it demands the communication channels to be a stream of some specific type. Though the former proposition allows reusability, the later proposition requires the knowledge about the ultimate use of the components. As an ad hoc tool to develop a parallel application, ASSIST may be beneficial, but it fails to prove its worthiness as a model / tool to design reusable components.

The ASSIST model is implemented on MPI as well as on Grid [37]. The PAS run-time system is built on top of LAM-MPI. In fact, LAM-MPI can exploit features of clusters and Grids. According to [1], most of the top super computers are built using commodity of the shelf hardware set and as a network of possibly multi-processor workstations. Note that MPI is an open standard and has been implemented for all major parallel computer systems. To cover the major user community, the authors of the COPS system reported a working run-time system on network clusters; at the same time their implementation conformed to the initial design of the run-time system for shared memory architectures. The use of expensive JavaSpace and costly synchronization methods is believed to prone to performance bottleneck. Moreover, the use of Jini technology makes the system not portable among different hardware platforms.

In this chapter, we discussed about the motivation behind SuperPAS. We categorized the users of the system and described their roles in the development process. We also described the model of the SuperPAS system and finally in the last section we compared this system with other closely related systems. In the next chapter, we describe the Skeleton Description Language (SDL) and some other implementation and design oriented issues.

Chapter 4

The Skeleton Description Language, Examples and Current Implementation

SuperPAS model is supported with a *Skeleton Description Language* (SDL) which is used by the skeleton designer to design abstract skeletons. An application developer may also use SDL to concretize an abstract skeleton directly. C++ code for a concrete skeleton is generated from the SDL code after concretization.

In this chapter, we discuss about the *Skeleton Description Language* (SDL). We describe different SDL constructs using examples. In Section 4.1, the SDL for abstract skeletons, targeted for the skeleton designer, is discussed. In the next section, we discuss about the added SDL constructs, targeted for the application developer, to concretize the abstract skeletons. Subsequently, we describe the design procedure for several skeletons in Section 4.3. In Section 4.4, some issues of the current implementation is highlighted. Finally, Section 4.5 demonstrates development of a parallel application using SuperPAS.

4.1 The SDL for Abstract Skeletons

In this section, we discuss about the grammar of the SDL, intended to design abstract skeletons. We also discuss several examples codes and elaborate them based on the discussion of the model, described in the previous chapter.

4.1.1 The Grammar

Following is the grammar of the SDL, written in the Backus-Naur Form (BNF). Note that, the start symbol of the grammar is abstract_skel.

```
abstract-skel          := [ param-decl ] [ pattern-alias-decl ]
param-decl             := param-type id-list ’;’
                        | param-type id-list ’;’ param-decl
param-type             := ’float’ | ’boolean’ | ’integer’
id-list                := identifier [ ’=’ expression ]
                        | identifier [ ’=’ expression ] ’,’ id-list
expression             := logical-OR-exp
                        | logical-OR-exp ’?’  expression ’:’  expression
logical-OR-exp         := logical-AND-exp
                        | logical-OR-exp ’||’ logical-AND-exp
logical-AND-exp        := internal-OR-exp
                        | logical-AND-exp ’&&’ internal-AND-exp
internal-OR-exp        := external-OR-exp
                        | internal-OR-exp ’|’ external-OR-exp
external-OR-exp        := AND-exp
                        | external-OR-exp ’^’ AND-exp
AND-exp                := equality-exp
                        | AND-exp ’&’ equality-exp
```

```
equality-exp          := relational-exp
                      |  equality-exp ’==’ relational-exp
                      |  equality-exp ’!=’ relational-exp
relational-exp        := shift-exp
                      |  relational-exp ’<’ shift-exp
                      |  relational-exp ’>’ shift-exp
                      |  relational-exp ’<=’ shift-exp
                      |  relational-exp ’>=’ shift-exp
shift-exp             := additive-exp
                      |  shift-exp ’<<’ additive-exp
                      |  shift-exp ’>>’ additive-exp
additive-exp          := multiplicative-exp
                      |  additive-exp ’+’ multiplicative-exp
                      |  additive-exp ’-’ multiplicative-exp
multiplicative-exp    := cast-exp
                      |  multiplicative-exp ’*’ cast-exp
                      |  multiplicative-exp ’/’ cast-exp
                      |  multiplicative-exp ’%’ cast-exp
cast-exp              := postfix-exp
                      |  unary-operator cast-exp
unary-operator        := ’+’ | ’-’ | ’~’ | ’!’
postfix-exp           := constant
                      |  identifier
                      |  identifier ’(’ argument-list ’)’
                      |  ’(’ expression ’)’
argument-list         := expression
                      |  argument-list ’,’ expression
```

```
constant               := integer-const | float-const | bool-const

bool-const             := ’false’ | ’true’

pattern-alias-decl     := pattern-decl

                       |  alias-decl

                       |  pattern-alias-decl pattern-decl

                       |  pattern-alias-decl alias-decl

pattern-decl           := ’pattern’ identifier ’(’ dimension ’)’

                          ’{’ pattern-desc ’}’

dimension              := expression

pattern-desc           := pattern-prop-desc ’;’

                       |  pattern-desc ’;’ pattern-prop-desc

pattern-prop-desc      := limit-desc | local-fnc-desc

                       |  init-fnc-desc | private-prim-desc

                       |  public-prim-desc | member-desc

limit-desc             := ’LIMITS’ ’=’ ’{’ limit-list ’}’

limit-list             := expression

                       |  limit-list ’,’ expression

local-fnc-desc         := ’LOCAL’ ’=’ ’{’ local-cpp-code ’}’

init-fnc-desc          := ’INITIALIZE’ ’=’ identifier

private-prim-desc      := ’PRIVATE’ ’=’ ’{’ private-cpp-code ’}’

public-prim-desc       := ’PUBLIC’ ’=’ ’{’ public-cpp-code ’}’

member-desc            := ’MEMBER’ ’=’ identifier

alias-decl             := ’alias’ ’{’ alias-desc ’}’

alias-desc             := alias-prop-desc

                       |  alias-desc ’;’ alias-prop-desc

alias-prop-desc        := alias-local-fnc-desc | alias-fnc-desc

alias-local-fnc-desc   := ’LOCAL’ ’=’ ’{’ alias-local-cpp-code ’}’
```

```
alias-fnc-desc          := 'RULE' '=' identifier
```

4.1.2 Discussion of the SDL

In this subsection, we discuss different language constructs through examples. While elaborating the examples, we put more emphasis on the semantics rather than the design procedure. The next section is entirely dedicated towards the design procedure of several skeletons.

```
integer size; // the parameter -> N
// the Wavefront pattern: abstract mapped space
pattern Wavefront(2) { // associated 2 dimensional VPG
   LOCAL      = {
      void init(void) { // the initialize function
         // set the dimensions
         for (int i = 0; i < GetDimension(); i++)
            SetDimensionLimit(i, size);
      }
      bool member(const Location & l) { // the membership function
         // l[0], l[1], ... indicate position of a node in a dimension
         //    in a row major order, i.e. l[0] is for the lowest
         //    dimension, l[1] is for next dimension, etc.
         if (l[1] <= l[0]) // row number <= column number
            return true;
         return false;
      }
   };
   INITIALIZE = init; // set the name of the initialization function
   MEMBER     = member; // set the name of the membership function
   PRIVATE    = { ... }; // private primitives
   PUBLIC     = { ... }; // public primitives
}
```

The above SDL code is for an abstract *Wavefront* skeleton. An Wavefront skeleton is an implementation of the Wavefront pattern. Figure 2, in the previous chapter, shows how the abstract topological space of a Wavefront pattern is mapped into a two dimensional VPG space. The figure also shows the final abstract mapped space.

As discussed in Subsection 3.4.3 of Chapter 3, the parameter *size* is declared at the beginning of the SDL description. The SDL supports three types of parameters: *boolean*, *integer* and *float*. The declaration of parameters is followed by the definition of abstract mapped spaces of the patterns in the skeleton. As Wavefront skeleton implements only one pattern, i.e. the Wavefront pattern, and hence the definition of that pattern follows the parameter declaration.

Table 1: Primitives available to initialize and membership functions

int GetDimension(void)	Returns the dimension of the pattern space
int GetDimensionLimit(int _dim)	Returns the boundary limit of *_dim*-th dimension
void SetDimensionLimit(int _dim, int _val)	Sets the the boundary limit of *_dim*-th dimension to *_val*

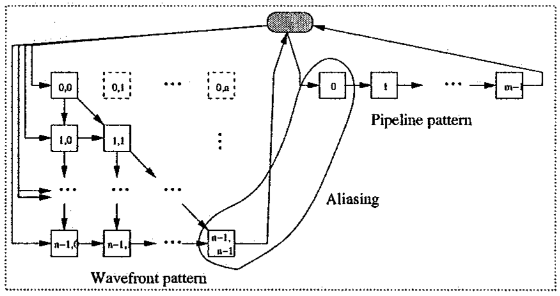
In the rules of the language construct, `pattern-decl`, `dimension` determines the dimension of required VPG. In the above code, the dimension of the VPG is chosen to be 2. The initialization and membership function are declared to be *init* and *member* through the SDL constructs `INITIALIZE` and `MEMBER` respectively. Both of these functions must be defined inside the `LOCAL` construct scope. The initialization function can be used to do validity checking of the values of parameters and boundary conditions of the dimensions of the VPG. The designer may impose some correctness constraints here. In the example SDL code, the *init* function sets the lengths of both of the two dimensions of VPG to *size*.

The membership function takes an address of a node, represented via the *Location* object, and returns whether that node is a non-null node. In the case of the Wavefront pattern, a node can be addressed as (i, j), where i and j are the row and column numbers respectively. Now, if $i \leq j$, the node is a non-null node, otherwise the node is null. In the SDL, it can found that the membership function returns true for all the nodes which satisfy the stated condition.

Several built-in functions are available to the initialization and membership function. All the three functions in Table 1 are available to the initialize function whereas only the first two are available to the membership function.

The code for `PUBLIC` and `PRIVATE` language construct are expanded below. Those two constructs represent the private and public primitives respectively. In the code, *SendToChildAt* and *RecvFromLastChild* are the private primitives. On the other hand, *SendLeft*, *RecvRight*, *SendRepresentative*, *IsAtDiagonal*, etc. are the public primitives of the pattern (and hence of the skeleton).

```
integer size; // the parameter -> N
// the Wavefront pattern: abstract mapped space
pattern Wavefront (2) { // associated 2 dimensional VPG
   LIMITS = ...;
   LOCAL = { ... };
   MEMBER = ...;
   // private primitives
   PRIVATE = {
      // send a message to a child located at <nRow, 0>
      bool SendToChildAt(int nRow, Msg & m) {
         Location l;
         l[0] = 0, l[1] = nRow;
         return SendChild(l, m);
      }
      // recv a message from the child located at <nSize - 1, nSize - 1>
      bool RecvFromLastChild(Msg & m) {
         Location l;
         l[0] = l[1] = nSize - 1;
         return RecvChild(l, m);
      }
   };
   // public primitives
   PUBLIC = {
      // COMMUNICATION PRIMITIVES
      // send from node <i, j> to <i + 1, j>
      void SendLeft(Msg &mm) {
         Location l = GetLocation();
         l[1] = l[1] + 1;
         SendPeer(l, mm);
      }
      // receive from node <i - 1, j> at <i, j>
      void RecvRight(Msg &mm) {
         Location l = GetLocation();
         l[1] = l[1] - 1;
         RecvPeer(l, mm);
      }
      // send from node <i, j> to <i, j + 1>
      void SendDown(Msg &mm) { ... }
      // receive from node <i, j - 1> at <i, j>
      void RecvUp(Msg &mm) { ... }
      // send from node <i, j> to <i + 1, j + 1>
      void SendDiagonalDown(Msg &mm) { ... }
      // receive from node <i - 1, j - 1> at <i, j>
      void RecvDiagonalUp(Msg &mm) { ... }
      // receive from the representative
      void RecvRepresentative(Msg &mm) {
         RecvParent(mm);
      }
      // send to the representative
      void SendRepresentative(Msg &mm) {
         SendParent(mm);
      }
      // STRUCTURAL PRIMITIVES
      // is at the first column
      bool IsAtFirstColumn(Msg &mm) {
         Location l = GetLocation();
         return l[0] == 0;
      }
      // is at the last row
      bool IsAtLastRow(Msg &mm) {
         Location l = GetLocation();
```



A composite skeleton

Figure 9: An aliasing example

```
         return l[1] == nSize - 1;
      }
      // is along the dialonal, i.e., <N - 1, *>
      bool IsAtDiagonal(Msg &mm) {
         Location l = GetLocation();
         return l[0] == l[1];
      }
      // is the last node, i.e., <N - 1, N - 1>
      bool IsLastNode(Msg &mm) {
         Location l = GetLocation();
         return l[0] == nSize - 1 && l[1] == nSize - 1;
      }
   };
}
```

It should be noted that the higher level pattern-specific primitives implemented by the designer are built on top of the basic primitives provided by the SDL. For example, the *RecvRight* primitive uses a built-in primitive: *RecvPeer.* Table 2 and Table 3 list some of the basic built-in primitives that can be used to construct higher level pattern-specific public and private primitives respectively.

Let us now concentrate on aliasing. Here, we revisit the example of Figure 7, which is redrawn in Figure 9 for better illustration. As can be seen in the figure, the lower-right most node of *Wavefront* pattern is aliased with the first stage of the *Pipeline* pattern. In the following SDL code, aliasing is performed through the *combine* function, specified through the RULE construct. The definition of the function is given inside LOCAL. The function performs the aliasing by calling the built-in function *AddAlias.* This function takes two nodes from two different abstract mapped spaces as arguments and aliases them.

```
integer Size, nStages;
```

Table 2: Some basic primitives available to public primitives

int GetDimension(void)	Returns the dimension of the pattern space
Location GetLocation(void)	Returns the address of the calling node in the VPG
bool SendPeer(Location &l, Msg &m)	Does a blocking send of message m to the peer with address l
Future ISendPeer(Location &l, Msg &m)	Does a non-blocking send of message m to the peer with address l
bool RecvPeer(Location &l, Msg &m)	Does a blocking receive of message m from the peer with address l
Future IRecvPeer(Location &l, Msg &m)	Does a non-blocking receive of message m from the peer with address l
bool BCastPeer(vector <Location> &vl, Msg &m)	Does a broadcast of message m to all the peers with any of the address in address vector vl
bool ScatterPeer(vector <Location> &vl, Msg vm)	Scatter messages. Message $vm[i]$ is sent to node with address $vl[i]$
bool GatherPeer(vector <Location> &vl, Msg vm)	Gather messages. Message $vm[i]$ is received from node with address $vl[i]$
bool BarrierPeer(vector <Location> &vl)	Barrier all the peers in the vl list include the calling nodes.
bool SendParent(Msg &m)	Does a blocking send of message m to the representative
bool ISendParent(Msg &m)	Does a non-blocking send of message m to the representative
bool RecvParent(Msg &m)	Does a blocking receive of message m from the representative
bool IRecvParent(Msg &m)	Does a non-blocking receive of message m from the representative
bool BarrierParent(Msg &m)	The child counter-part for the function *BarrierChild*

Table 3: Some basic primitives available to private primitives

int GetDimension(void)	Returns the dimension of the pattern space
bool SendChild(Location &l, Msg &m)	Does a blocking send of message m to the child with address l
Future ISendChild(Location &l, Msg &m)	Does a non-blocking send of message m to the child with address l
bool RecvChild(Location &l, Msg &m)	Does a blocking receive of message m from the child with address l
Future IRecvChild(Location &l, Msg &m)	Does a non-blocking receive of message m from the child with address l
bool BCastChildren(vector <Location> &vl, Msg &m)	Does a blocking send of message m to all the children with any of the address in the address vector vl
bool ScatterChildren(vector <Location> &vl, Msg vm)	Scatter messages in message vector vm. Message $vm[i]$ is sent to the child with address $vl[i]$
bool GatherChildren(vector <Location> &vl, Msg vm)	Gather messages in message vector vm. Message $vm[i]$ is received from the child with address $vl[i]$
bool BarrierChildren(vector <Location> &vl)	Barrier all children in the address vector vl with the parent node

```
// the Wavefront pattern
pattern Wavefront(2) {
   ...
}
pattern Pipeline(1) {
   ...
}
alias {
   LOCAL =
      void combine(void) {
         Location lWF, lPipe;
         // the lower-rightmost node of wavefront
         lWF[0] = Wavefront.GetDimensionLimit(0) - 1;
         lWF[1] = Wavefront.GetDimensionLimit(1) - 1;
         // the first stage of pipeline
         lPipe[0] = Pipeline.GetDimensionLimit(0) - 1;
         AddAlias(&Wavefront, lWF, &Pipeline, lPipe);
      }
   }
   RULE = combine;
}
```

4.2 The SDL for Concrete Skeletons

In this section, we describe the additional grammar of the SDL, required to concretize abstract skeletons. We also discuss the grammar with an example.

4.2.1 The Grammar

The grammar of the SDL for concrete skeletons in BNF form is shown in the following. Besides, it introduces some new symbols and rules. The start symbol of the grammar concrete-skel. Complete SDL grammar is composed of the following grammar together with the grammar discussed in subsection 4.1.1

Refer to Subsection 4.1.1, if rules to expand a non-terminal is omitted in the grammar below. Note that, rules mentioned here prevails over the rules mentioned in the last section.

```
concrete-skel              := [ param-decl ] [ pattern-alias-label-decl ]
pattern-alias-label-decl := pattern-alias-decl | label-decl
id-list                    := identifier '=' expression
```

```
                            | identifier '=' expression ',' id-list
label-decl                  := 'label' '=' '{' label-desc '}'
label-desc                  := label-local-fnc-desc | label-fnc-desc
                            | label-fnc-desc | label-local-fnc-desc
local-fnc-desc              := 'LOCAL' '=' '{' label-cpp-code '}'
label-fnc-desc              := 'RULE' '=' identifier
```

4.2.2 Discussion of the Grammar

The concretization procedure is top-down. The application developer chooses appropriate skeletons from the repository of abstract skeletons. Then she bounds the parameters of the skeleton with appropriate values and decides about the type of the children of the skeleton. Defining the type of children results in construction of skeleton hierarchy.

The concrete skeleton CS_{N_i} of the abstract skeleton AS_{N_i} can be expressed with the set $\{PV(N_i), LS(N_i)\}$. If *prm* is a parameter of the skeleton AS_{N_i}, $(prm, val_i) \in PV(N_i)$, where parameter *prm* is bounded with the value val_i; and if l is a non-null node in the skeleton space of the skeleton, $(l, AS_{M_k}) \in LS(N_i)$, where l is labeled with AS_{M_k}, the k-th instance of the abstract skeleton AS_M. Note that this definition results in a recursive and top-down concretization procedure. Each of the AS_{M_k}, used to label the children of CS_{N_i}, is required to be concretized unless AS_M is an abstract *Singleton* skeleton.

Let us consider the skeleton hierarchy of Figure 10(a), decided by an application developer. According to her decision the first stage of the *Pipeline* is labeled with the MIS (*Master and Identical Slaves*) skeleton. The second and the third stages are labeled with DP (*Data Parallel*) and MNIS (*Master and Non-Identical Slaves*) skeletons respectively.

The above hierarchy can be expressed in a LISP expression as shown below.

```
myPipeline (  ; myPipeline is an instance of Pipeline skeleton
   3,         ; set number of stages to three
   myMIS,     ; the first stage is labeled with an instance of MIS skeleton
   myMIS,     ; the second stage is labeled with an instance of DP skeleton
   myMINS     ; the third stage is labeled with an instance of MNIS skeleton
)
```

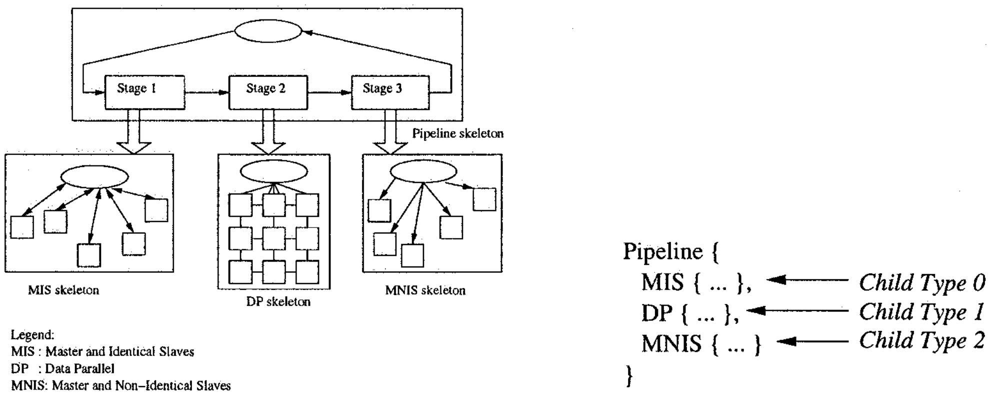
Now say, the developer decides to have at most 10 slaves for the *myMIS* skeleton and to label each of the slaves with *myFDSingleton*, an instance Singleton skeleton. Here the purpose of *myFDSingleton* is to detect faces in a picture. After this refinement, the above expression can be refined to following.

```
myPipeline (  ; myPipeline is an instance of Pipeline skeleton
   3,         ; set number of stages to three
   myMIS(     ; the first stage is labeled with an instance of MIS skeleton
      10,           ; 10 slaves
      myFDSingleton ; slaves are of myFDSingleton
   ),
   myMIS,     ; the second stage is labeled with an instance of DP skeleton
   myMINS     ; the third stage is labeled with an instance of MNIS skeleton
)
```

The SDL implementation simulates same concepts with three components: (1) an expression for the hierarchy, (2) expressions to set the values of each of the skeleton parameters, and (3) a labeling functions that labels each of the non-null node of the skeleton with a skeleton in the next level of the hierarchy. The SDL expression of the example hierarchy of Figure 10(a) is shown in Figure 10(b).

In the expression for the hierarchy, the developer is specifying the names of the skeletons which are available to become the children of the concrete Pipeline skeleton. Beside the hierarchy, she also needs to mention that MIS skeleton (child type 0) should take the place of the first stage whereas DP (child type 1) and MNIS (child type 2) should take the place of the second and third stage respectively. This mapping is expressed in SDL through the following *label* code block.

```
integer Stages = 3; // bind the parameter: 3 stages
// the pipeline pattern
pattern Pipeline(1) {
   ...
}
label {
   LOCAL = {
      void labelChildren(void) {
         Location l;
         for (int i = 0; i < Pipeline.GetDimensionLimit(0); i++) {
```



(a) Pictorial View of Hierarchy

(b) Hierarchy in SuperPAS

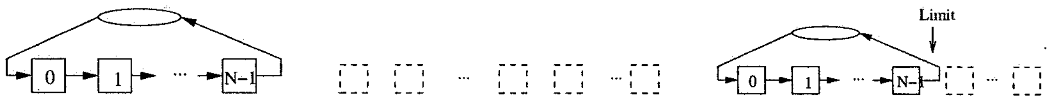
Figure 10: One level of skeleton hierarchy

```
10              l[0] = i;
11              AddLabel(&Pipeline, l, i);
12          }
13      }
14    };
15    RULE = labelChildren;
16 }
```

In the above code, the labeling function, *labelChildren* (specified through RULE construct), labels each of the children of Pipeline skeleton with a type, i.e. with a concrete skeleton. Type of node i (line 10) of the abstract mapped space of the Pipeline pattern is also i (the third argument of *AddLable* at line 11). After concretizing the Pipeline skeleton, the developer needs to decide about concretizing each of the children skeleton of the Pipeline (i.e., DP, MIS, MNIS) and then the children of those children and so on, until she reaches to the leaves with instances of the *Singleton* skeletons, which has no back-end and hence no children.

4.3 Example Design of Abstract Skeletons

In this section, we describe the design procedure of several abstract skeletons. The designed skeletons are *Pipeline*, *Mesh*, *Cube-Connected-Cycles*, *X-Tree* and *Singleton*



(a) The pipeline pattern

(b) One dimensional VPG

(c) The abstract mapped space

Figure 11: Designing the abstract Pipeline skeleton

skeleton.

4.3.1 Abstract Pipeline Skeleton

In Figure 11(a), the structure of the Pipeline pattern is shown. A Pipeline pattern is composed of the representative and N stages: stage 0 to stage $N-1$. Each stage receives a problem from the previous stage, computes the solution and sends the result to the next node. The first and last stages are little bit exception of this rule, where the first stage receives the problem from the representative and the last stage forwards the solution back to the representative.

The structure of the stages suggests that an one dimensional VPG is sufficient to map the stages very well. In Figure 11(b) an one dimensional VPG is shown. Finally, in Figure 11(c), the mapping of the Pipeline pattern on top of the VPG is shown.

The number of stages in the Pipeline is a design choice and hence a parameter. The number of stages can be represented with an integer expression and in this example is represented by the parameter *nStages*. As shown in Figure 11(c), the trivial way of mapping the pattern onto the VPG is to map the stage i of the Pipeline pattern on top of the VPG node with address $< i >$. As a result, node with address $< 0 >$ is the place holder for the first stage whereas $< nStages - 1 >$ is the place holder for the last stage.

The private primitives for this skeleton are the primitives for the representative:

(1) *send to the first stage* i.e., the representative sends the problem to the node with address $< 0 >$ and (2) *receive from the last stage* i.e., the representative receives the solution computed by the node with address $< nStages - 1 >$.

For node $< i >$, except the node for the last stage, a primitive *send to the next stage* means a send to node $< i{+}1 >$. For the last stage the same primitive means *send to the representative.* In similar way, the *receive from the previous stage* primitive can be defined. Those two primitives are the public primitives for the Pipeline skeleton.

Finally, the SDL code for the abstract Pipeline skeleton is given bellow.

```
// File name: pipeline.skel
// The Pipeline skeleton
integer nStages;
// Pipeline pattern inside Pipeline skeleton
pattern pipeline(1) {
   LIMITS = {nStages};
   LOCAL = {
      // everybody is non-null after setting the limit
      bool members(const Location &l) {
         return true;
      }
   };

   // returns the members in the pattern space
   MEMBER=members;
   // private primitives
   PRIVATE= {
      // receive from last stage
      void RecvFromLastStage(Msg &mm) {
         Location l;
         l[0] = nStages - 1;
         RecvChild(l, mm);
      }
      // send to the first stage
      void SendToFirstStage(Msg &mm) {
         Location l;
         l[0] = 0;
         SendChild(l, mm);
      }
   };
   // public primitives
   PUBLIC= {
      // receive from previous stage
      void RecvPrev(Msg &mm) {
         Location l = GetLocation();
         if (l[0] == 0) { // if first stage
            RecvParent(mm);
         } else { // otherwise
            l[0] = l[0] - 1;
            RecvPeer(l, mm);
         }
      }
      // send to next stage
      void SendNext(Msg &mm) {
         Location l = GetLocation();
         if (l[0] == nStages -  1) { // if last stage
```

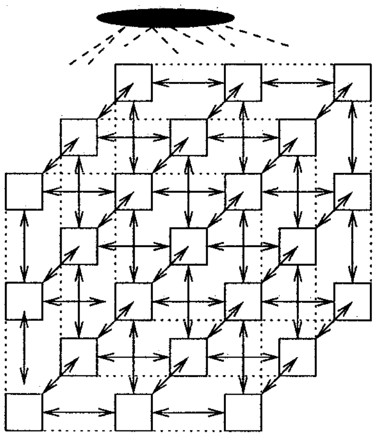


Figure 12: A Mesh pattern

```
                SendParent(mm);
            } else { // otherwise
                l[0] = l[0] + 1;
                SendPeer(1, mm);
            }
        }
    };
}
```

4.3.2 Abstract Mesh Skeleton

The Mesh pattern is one of the most common pattern in parallel computing. It represents the data-parallel paradigm of parallel computing. This pattern is also known as the *Cube* pattern. The *Mesh* skeleton represents a generic k-dimensional Mesh pattern. The application developer decides about the proper dimension for her Mesh pattern, according to the needs of the application and hence k is a parameter. Figure 12 shows a three dimensional Mesh pattern.

A node of the pattern is addressed with a k-tuple integer: $(a_{k-1}, \ldots, a_1, a_0)$, where a_i is the position of that node in the i-th dimension. Each node has a neighbor in each direction. A direction is represented by the vector $[d_{k-1}, \ldots, d_1, d_0]$, where $d_i \in \{-1, 0, +1\}$ for $0 \leq i < k$ and $d_i \neq 0; \exists i$. The neighbor of $(a_{k-1}, \ldots, a_1, a_0)$ in the direction $[d_{k-1}, \ldots, d_1, d_0]$ is $(a'_{k-1}, \ldots, a'_1, a'_0)$, where $a'_i = a_i + d_i$ for $0 \leq i < k$.

There is a choice to make about the neighboring nodes of the nodes at the edges of the Mesh. The first choice is not to have a neighbor in a direction that results in an address, out of the abstract mapped space. The second choice is to always have a neighbor node. In this case, all the edges are considered to be wrapped around. A little thinking will reveal that this choice about neighboring node is also a parameter for this pattern.

In case of data-parallel computation, an parallel computational node of the Mesh pattern usually share (communicate) information between neighbors. This communication is certainly the public primitives. The representative usually partitions the problem into sub-problems and scatters the sub-problems among the parallel computational nodes. At the end of the computation, the parallel computational nodes returns the result back to the representative. The representative gathers the solutions of the sub-problems and merges them to have the global solution. This description of the Mesh pattern defines the private and public primitives. The abstract skeleton description in SDL is given below.

```
// File name: mesh.skel
// The Mesh skeleton
integer k; // k-dimensional Mesh
bool fWrapping; // wrapping at the edge
// Mesh pattern inside Mesh skeleton
pattern Mesh (k) {
   // application developer should define LIMITS
   // otherwise they will take default values
   // LIMITS = { ... };
   LOCAL = {
      bool is_member(const Location & l) {
         return true;
      }
   };
   MEMBER = is_member;
   PRIVATE = {
      void GenerateLocation(vector <Location> &vl, Location &l, int dim) {
         if (dim < 0) { // base condition
            vl.push(l);
         } else {
            for (int i = 0; i < GetDimensionLimit(dim); i++) {
               l[dim] = l[dim] + 1;
               GenerateLocation(vl, l, dim - 1);
            }
         }
      }
      // scatter message to children in row major order
      bool ScatterToChildren(Msg * vm) {
         // prepare the receiver vector
         vector <Location> vl;
         Location l;
         GenerateLocation(vl, l, GetDimension() - 1);
         return ScatterChildren(vl, vm);
      }
      // Gather from Children in row major order
      bool GatherFromChildren(Msg * vm) {
         // prepare the receiver vector
         vector <Location> vl;
         Location l;
         GenerateLocation(vl, l, GetDimension() - 1);
         return GatherChildren(vl, vm);
      }
   };
   PUBLIC = {
      // COMMUNICATION PRIMITIVES
      // send to a neighbor towards a direction
      void SendNeighbor(int * direction, Msg &mm) {
         Location l = GetLocation();
         for (int i = 0; i < k; i++) {
            int deviation = direction[i] == 0 ? 0 : direction[i] < 0 ? -1 : 1;
            l[i] = l[i] + deviation;
            if (fWrapping)
               l[i] = (l[i] + GetDimensionLimit(i)) % GetDimensionLimit(i);
         }
         ISendPeer(l, mm);
      }
      // receive from a neighbor towards a direction
      void ReceiveNeighbor(int * direction, Msg &mm) {
         Location l = GetLocation();
         for (int i = 0; i < k; i++) {
            int deviation = direction[i] == 0 ? 0 : direction[i] < 0 ? -1 : 1;
            l[i] = l[i] + deviation;
            if (fWrapping)
               l[i] = (l[i] + GetDimensionLimit(i)) % GetDimensionLimit(dim);
         }
         RecvPeer(l, mm);
      }
      void RecvRepresentative(Msg &mm) {
         RecvParent(mm);
      }
      // send to the representative
      void SendRepresentative(Msg &mm) {
         SendParent(mm);
      }
      // STRUCTURAL PRIMITIVE
      // is a node at the begining edge of dimension 'dim'
      bool IsAtBeginning(int _dim) {
         Location l = GetLocation();
         return l[dim] == 0;
      }
      // is a node at the ending edge of dimension 'dim'
      bool IsAtEnding(int _dim) {
         Location l = GetLocation();
         return l[dim] == GetDimensionLimit(dim) - 1;
      }
   };
}
```

4.3.3 Abstract Cube-Connected-Cycles Skeleton

Cube-Connected-Cycles pattern (CCC) [31, 38] is an interesting pattern. A k dimensional CCC consists of $k \times 2^k$ computational nodes. A CCC is k dimensional

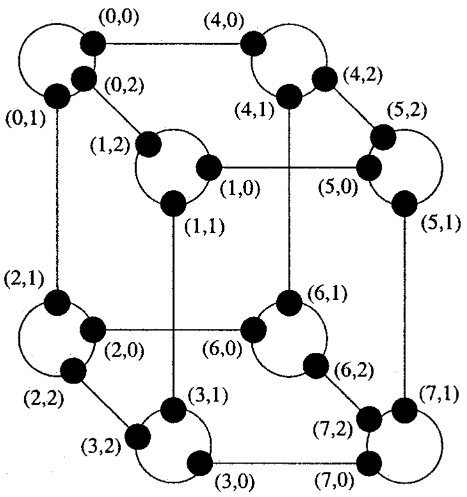


Figure 13: The Cube-Connected-Cycles pattern

cube where each corner of the cube is a ring of k elements. In Figure 13, a three dimensional CCC is shown.

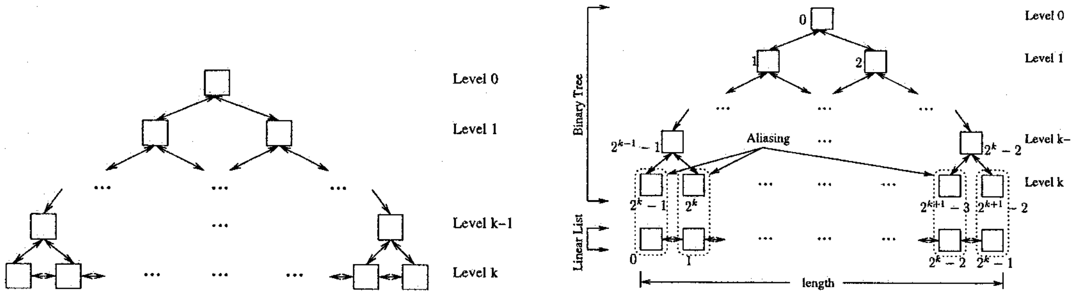
Each computational node of the pattern can be addressed with two integers (i, j), where $2^k < i \leq 0$ and $k < j \leq 0$. Neighbors of (i, j), in the same ring (i.e., the i-th ring), are (i, m) and (i, n), where $m = (j + 1) \bmod k$ and $j = (n + 1) \bmod k$. Node (i, j) has another neighbor in p-th ring with address (p, j), where $p = i \oplus 2^{k-j}$, i.e. p is calculated by inverting the j-th most significant bit of i.

We already know that each of the nodes in the CCC can be addressed using two tuple integers (i, j). The ranges of value i and j can take are also known. So, each of the nodes of the CCC can be represented by a node in a two dimensional space and node (i, j) of the CCC can be addressed as $< i, j >$ in that two dimensional space. As the discussion suggest, the number of columns and rows of the VPG will be bounded to k and 2^k respectively. Now, constructing the primitives is simple as both of the $\mathcal{M}$ and $\mathcal{T}$ are known.

```
// File name: sccc.skel
// the CCC skeleton
integer k;
// the abstract mapped space for CCC pattern
pattern ccc (2) {
   LIMITS = {k, integer(pow(k, 2))};
   LOCALS = {
      bool mem (const Location &l) {
         return true;
      }
   };
   MEMBER = mem;
   PRIVATE = {
      bool SendCCCNode(int x, int y, Msg &m) {
         Location l;
         l[0] = x, l[1] = y;
         return SendChild(l, m);
      }
      bool RecvCCCNode(int x, int y, Msg &m) {
         Location l;
         l[0] = x, l[1] = y;
         return RecvChild(l, m);
      }
   };
   PUBLIC = {
      // for node <i, j>
      // send to node <i, (j + 1) mod k>
      bool SendNextNodeInRing(Msg &m) {
         Location l = GetLocation();
         l[0] = (l[0] + 1) % k;
         SendPeer(l, m);
      }
      // send to node <i, (j + k - 1) mod k>
      bool SendPrevNodeInRing(Msg &m) {
         Location l = GetLocation();
         l[0] = (l[0] + k - 1) % k;
         SendPeer(l, m);
      }
      // send to node <i XOR 2^(k - j), j>
      bool SendNextNodeOutRing(Msg &m) {
         Location l = GetLocation();
         l[1] = (l[1] ^ int(pow(2, (k - l[0]))));
         SendPeer(l, m);
      }
      // receive from node <i, (j + 1) mod k>
      bool RecvNextNodeInRing(Msg &m) {
         Location l = GetLocation();
         l[0] = (l[0] + 1) % k;
         RecvPeer(l, m);
      }
      // receive from node <i, (j + k - 1) mod k>
      bool RecvPrevNodeInRing(Msg &m) { ... }
      // receive from node <i XOR 2^(k - j), j>
      bool ReceiveNextNodeOutRing(Msg &m) { ... }
   };
}
```

4.3.4 An Abstract X-Tree Skeleton

In this subsection, we shall design an abstract X-Tree [39, 40] skeleton by composing a *Binary Tree* and a *Linear List* skeleton. The simplest version of the X-Tree pattern is shown in Figure 14(a). In the pattern, each leaf of a binary tree is connected to the neighboring leaves.



(a) The X-Tree pattern

(b) Composing Binary Tree and Linear List skeletons

Figure 14: Designing X-Tree skeleton through composition

Say, we already have a *Binary Tree* and a *Linear List* skeleton in the repository. Those skeletons implement the Binary Tree and Linear List pattern respectively. The abstract mapped space of both of the skeletons are built on-top of two one dimensional VPG. In Figure 14(b), the number shown beside each of the parallel computational nodes indicates the address of the node where it is mapped onto.

As shown in the figure, the height of the tree in the Binary Tree pattern is a design choice, i.e. a parameter for the Binary Tree skeleton. Similarly, the length of the list is a parameter for the Linear List skeleton. Let us assume that in SDL code those two parameters are represented as *height* and *length.*

The Figure 14(b) also shows the aliases required to compose two skeletons to build the targeted X-Tree skeleton. If the height of the tree is $k+1$, i.e. the highest level is k, the leaves would have addresses from 2^k-1 to $2^{k+1}-2$. To compose a Linear List with a Binary Tree, a Linear List must be of length $2^k = (2^{k+1}-2)-(2^k-1)+1$. Node i of the abstract mapped space of Binary Tree should be aliased with node j of the abstract mapped space of Linear List, where $i = 2^k - 1 + j$ and $0 \leq j < 2^k$. Following SDL code reflects this idea.

```
integer height; // from Binary Tree skeleton
integer length = integer(pow(2, height - 1)); // if height=k+1, length=2^k
// the Binary Tree pattern
```

```
pattern BinaryTree(1) {
   ...
}
pattern LinearList(1) {
   ...
}
alias {
   LOCAL =
      void doAlias(void) {
         for (int j = 0; j < length; j++) { // 0 <= j < 2^k
            Location lBT, lLL;

            // for node from binary tree: 2^k - 1 + j = length - 1 + j
            lBT[0] = length - 1 + j;
            // for node from linear list: j
            lLL[0] = j;
            AddAlias(&BinaryTree, lBT, &LinearList, lLL);
         }
      }
   }
   RULE = doAlias;
}
```

4.3.5 Abstract Singleton Skeleton

Abstract Singleton skeleton is probably the simplest skeleton. It does not have any back-end and hence no parameters and no primitives. As a result, an abstract Singleton skeleton has no SDL code, as shown below.

```
// File name: singleton.skel
// the Singleton skeleton -> it has no back-end and
//    hence no abstract mapped spaces and primitives
```

4.4 The Current Implementation

The current implementation of the SuperPAS run-time system is written on top of LAM-MPI [41] version 7.0, which is an MPI-2 implementation. Though the system is not tested, we believe that it will also work well on any other MPI-2 implementations. The translator that translates concrete skeletons in SDL into C++ code is written using Perl [42]. The translator is built on top of a recursive descend LL1 parser. The SuperPAS tools use features supported by the POSIX compatible operating systems. The system also uses standard *NIX development tools like *make*, *GNU* [43] C++ compiler, etc.

The implementation supports the SuperPAS model with fusion aliasing only. Instantiation of an abstract skeleton means copying the file of the abstract skeleton in SDL to the application project directory. The composition is performed by copying SDL sources from more than one abstract skeletons into a new file.

The library object *Msg* is a universal container to encrypt any type of messages. Data structure a message, to be sent / receive, should be represented by an object, inherited from *Msg*. The user must overwrite two methods of that inherited object: *Marshal* and *Unmarshal*. In fact, in those two methods, the user specifies what composes the message and how to send and receive each component of the message.

Say, we like to have an object representing a gray scale image. A gray scale image has three properties: (1) height, (2) width and (3) gray scaled pixel data. The following *MsgImage* object is a candidate object to represent such an image. As the object is inherited from the *Msg* object and as the methods *Marshal* and *Unmarshal* are overwritten according to the need, the object can be used as argument to any of the communication functions, provided by the SuperPAS run-time system.

```
class MsgImage : public Msg {
   int width, height;
   int * data;
public:
   MsgImage(void) : Msg(), width(0), height(0), data(NULL) { }
   MsgImage(int _height, int _width) : Msg(), height(_height), width(_width) {
      data = ...;
   }
   // from gd image library
   MsgImage(const gdImagePtr im) : Msg(), height(gdImageSY(im)), width(gdImageSX(im)) {
      // copy the data from gd library
      ...
   }
   void SetImage(const gdImagePtr im) { ... }
   int GetWidth(void) { return width; }
   // other methods
   ...
   // FOLLOWING METHODS MUST BE OVER WRITTEN
   // to marshal this object
   void Marshal(void) {
      // marshal width
      MarshalData(width);
      // marshal height
      MarshalData(height);
      // marshal image data
      MarshalData(data, width * height);
   }
   // to unmarshal this object
   void Unmarshal(void) {
      // unmarshalling must be in same order of marshalling
```

```
        // unmarshal width
        UnmarshalData(width);
        // and height
        UnmarshalData(height);
        if (data) delete []data;
        data = new int[width * height];
        // we have proper memory, now unmarshal image data
        UnmarshalData(data, height * width);
    }
};
```

Representative of a skeleton access the private primitives directly, whereas, it accesses the public primitives (i.e. primitives of the parent skeleton) through a special instance of an object, named *External.* So, *External.SendNext(...)* can be used to send message to the next stage (assuming that Pipeline is the parent in the skeleton hierarchy). Note that, the root skeleton of the hierarchy has an *External* object with no methods available and leaves of the hierarchy (i.e instances of *Singleton* skeleton) do not have any private primitive available for use.

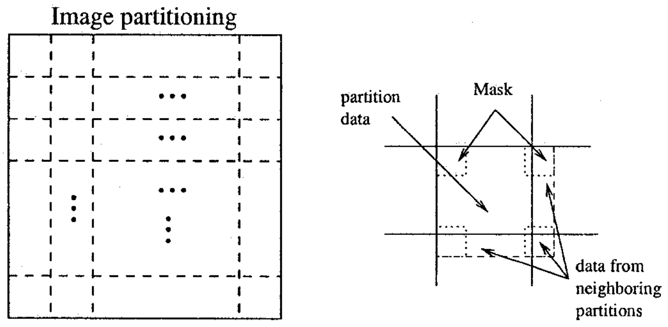
4.5 An Image Convolution Application

Image convolution is an important application in the domain of image processing [44]. Here we describe the step by step procedure to develop a parallel image convolution application using SuperPAS.

4.5.1 Problem Description

In an image convolution application, a mask is applied to each of the pixels of the image. The simplest way to make the procedure parallel is to divide the whole image into several parts (into columns and rows) and distribute different parts to different processes and each process computes the convolution of the part of the image assigned to it. The idea is shown in Figure 15(a).

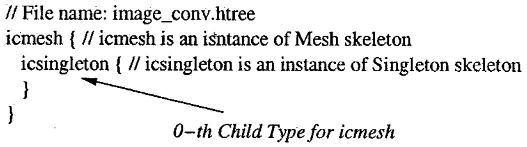
Unfortunately, there are dependencies among those computing processes. Each process needs to have some extra data from the neighboring processes as shown in



(a) Dividing image

(b) Data dependency

Figure 15: Design of an image convolution application



```
// File name: image_conv.htree
icmesh { // icmesh is an isntance of Mesh skeleton
  icsingleton { // icsingleton is an instance of Singleton skeleton
  }
}
```

0-th Child Type for icmesh

Figure 16: The first level skeleton hierarchy for an image convolution application

Figure 15(b). Hence neighbors should communicate and exchange some data with each others before starting the actual convolution operation.

4.5.2 Concretizing Abstract Skeletons

As the problem description suggests, our application demands a two dimensional Mesh skeleton to be used. In the skeleton the representative will read the image and mask file and distribute them among the children nodes. As for the children nodes the convolution task is a sequential computation, the Singleton skeleton would the perfect candidate. This first level hierarchy for this application is shown in Figure 16.

The Concordia Beowulf Cluster [45] consists of 10 dual-processor slave nodes. So, for better performance, we choose to have 20 parallel computing processes, doing the convolution. For input images size of 2048 × 1536 (2048 columns and 1536 rows) we may choose to divide the images among 5 × 4 children. Based on this decision, we can concretize the Mesh skeleton into *icmesh* concrete skeleton. The code for this concrete skeleton is shown in the followings:

```
// File name: icmesh.skel
// An instance of Mesh skeleton is created by copying mesh.skel to icmesh.skel
//   and performing necessary updates for concretization
// The icmesh skeleton
integer k = 2; // binding the parameter: two dimensional VPG
bool fWrapping = false; // no wrapping at the edge
// Mesh pattern inside Mesh skeleton
pattern Mesh (k) {
   LIMITS = {4, 5}; // 4 columns and 5 rows
   ...
}
```

We also need to specify a labeling function to label each children of the *icmesh* skeleton with *icsingleton*, an instance of the abstract Singleton skeleton. The SDL code block for this labeling is added at the end of the above skeleton definition.

```
// File name: icmesh.skel
...
pattern Mesh (k) {
   ...
}

label {
   LOCAL = {
      void GenerateLocation(vector <Location> &vl, Location &l, int dim) {
         if (dim < 0) { // base condition
            vl.push(l);
         } else {
            for (int i = 0; i < GetDimensionLimit(dim); i++) {
               l[dim] = l[dim] + 1;
               GenerateLocation(vl, l, dim - 1);
            }
         }
      }
      void label(void) {
         vector <Location> vl;
         Location l;
         // generate all possible children locations
         GenerateLocation(vl, l, GetDimension() - 1);
         // for all children ....
         for (int i = 0; i < vl.size(); i++) {
            // label with icsingleton (0-th child type)
            AddLabel(&mesh, vl[i], 0);
         }
      }
   };
   RULE = label;
}
```

Now at the second level of the hierarchy, concretizing the *icsingleton* skeleton, an instance of Singleton skeleton, needs no binding of parameters and no labeling of children, as the Singleton skeleton does neither have any parameter nor have any back-end. The resulting SDL code for the concretized *icsingleton* skeleton is as follows:

```
// File name: icsingleton.skel
// An instance of Singleton skeleton is created by copying singleton.skel
//    to icsingleton.skel
// the Singleton skeleton -> it has no back-end and hence no pattern spaces
//    and primitives
```

4.5.3 Code-Complete Modules

Before writing the code complete modules, the developers needs to generate the C++ code for the skeleton hierarchy, she has developed, using the supported tools. The tools will generate one C++ file for each of the skeletons having the same name as the concrete skeleton.

The developer only needs to fill-up the code for the representative of each skeleton. The C++ code for each skeleton contains an object, called *Skeleton*. The *Rep* method of each of the *Skeleton* object is interpreted as the representative of the corresponding skeleton.

In the image convolution application, the *icsingleton.cpp* and *icmesh.cpp* C++ files are for *icsingleton* and *icmesh* skeleton respectively. Here, we will use the *MsgImage* object, defined in the last section, for communication purpose. The code-complete *icsingleton* module would look like as follows.

```
...
class Skeleton : ... {
   ...
public:
   Skeleton(...) : ... { }
   // NEWLY ADDED METHODS (BY DEVELOPER) BEGINS
   void RecvRight(Msg &m) {
      static int * p = {+1, 0}; // right node in a 2-d mesh
      External.RecvNeighbor(p, m);
   }
   void RecvDown(Msg &m) {
      static int * p = {0, +1}; // down node in a 2-d mesh
      External.RecvNeighbor(p, m);
   }
   void RecvDiagonal(Msg &m) {
      static int * p = {+1, +1}; // diagonal node in a 2-d mesh
      External.RecvNeighbor(p, m);
   }
   void SendLeft(Msg &m) {
      static int * p = {-1, 0}; // left node in a 2-d mesh
      External.SendNeighbor(p, m);
   }
   void SendUp(Msg &m) {
      static int * p = {0, -1}; // up node in a 2-d mesh
      External.SendNeighbor(p, m);
```

```
}
void SendDiagonal(Msg &m) {
   static int * p = {-1, -1}; // diagonal node in a 2-d mesh
   External.SendNeighbor(p, m);
}
bool IsAtFirstColumn(void) {
   return External.IsAtBeginning(0);
}
bool IsAtLastColumn(void) {
   return External.IsAtEnding(0);
}
bool IsAtFirstRow(void) {
   return External.IsAtBeginning(1);
}
bool IsAtLastRow(void) {
   return External.IsAtEnding(1);
}
// ADDED METHODS ENDS
void Rep(void) {
   // Fill in with your code
   MsgImage imageIn, imgi, mask; // To receive the main image and the mask
   MsgImage sm[3]; // to send to neighbor
   MsgImage rm[3]; // to receive from neighbor

   // receive the mask
   External.RecvRepresentative(mask);
   // receive the part of the image from parent
   External.RecvRepresentative(imageIn);

   // send to the neighbors
   if (!IsAtFirstColumn()) {
      // prepare sm[0] to send to the left node
      SendLeft(sm[0]);
   }

   if (!IsAtFirstRow()) {
      // prepare sm[1] and sm[2] to send to
      // the up and diagonal nodes respectively
      SendUp(sm[1]);
      SendDiagonal(sm[2]);
   }

   // receive from the neighbor
   if (!IsAtLastColumn()) {
      // prepare rm[0] to receive from the right node
      RecvRight(rm[0]);
   }

   if (!IsAtLastRow()) {
      // prepare rm[1] and rm[2] to receive from
      // the down and diagonal nodes respectively
      RecvDown(rm[1]);
      RecvDiagonal(rm[2]);
   }

   // combine rm[0..3] and imageIn in imgi
   ...
   // now convolute
   MshImage imgo(imgi.dx(), imgi.dy());
   Convolute(imgi, mask, imgo);

   // now compute imgOut from imgo
   // imgOut contains only that part of imege that is required to send
   MsgImage imageOut(imageIn.dx(), imageIn.dy());
   ...
```

```
      // now send the result
      External.SendRepresentative(imageOut);
   }
};

int main(void) {
   ...
}
```

The code-complete *icmesh* module would look like as follows.

```
...
   void Rep(void) {
      // Fill in with your code
      MsgImage imgMain, mask;
      // read the image from file into imgMain object
      ...
      // and the mask in mask object
      ...
      // now partition the image
      int nParts = GetDimensionLimits(0) * GetDimensionLimits(1);
      MsgImage * imgParts = new MsgImage[nParts];
      ... // and divide imgMain among imgParts
      // now send to the children
      ScatterToChildren(imgParts);

      // now gather the convoluted image parts from children
      GatherFromChildren(imgParts);

      // combine back the main image from parts in the imgMain object
      ...
      // and write the result back to a file
      ...
   }
...
```

This completes the development procedure. The developer may now compile and run the application.

In this chapter, we described the implementation of the SuperPAS model and elaborated through several examples. In the next chapter, we discuss about some usability and performance related issues.

Chapter 5

Usability and Performance Issues

This chapter is divided into two sections. In the first section, we address the issue of usability of SuperPAS. We also describe the conducted experiments and results. In the second chapter, we address the issue of performance of applications developed using SuperPAS.

5.1 Usability

The main reason of introducing PAS and in-turn SuperPAS is to have a more usable, pattern-based parallel programming system. In this section, the usability issues of SuperPAS are elaborated.

To measure the usability of the SuperPAS system, we conducted an experiment on a group of graduate students. We elaborate our experiment environment, results and conclusions in the following three subsections.

5.1.1 Environment

To conduct the usability experiment, we choose a group of twelve graduate students. At the time of the experiment all the students in the test group were enrolled in a introductory parallel and distributed systems course. We assumed that as graduate students they have sufficient background to learn, adapt and use new systems. In the specified introductory course, the students were introduced to the different parallel hardware systems as well as different parallel programming models and environments.

At first, students were introduced to the MPI (Message Passing Interface) model of parallel programming through a tutorial of one hour. Then they were asked to develop a parallel image convolution application on the Concordia Beowulf Cluster. As the students had no background of programming with MPI, the total development time was divided into two parts: (1) the *learning time* to learn details (beyond the tutorial) about the MPI environment and (2) the *coding and debugging time* to write the code for the application and debug it, if necessary.

Later on, an one hour tutorial was arranged to introduce the group to the SuperPAS model and to SDL. Then the students were asked to develop the same application, but this time using SuperPAS. Students were asked to divide their efforts into three parts: (1) the *learning time* to learn about the SuperPAS and SDL, (2) the *skeleton design time* to design the required abstract skeletons and (3) the *application development time* to concretize the required skeletons and fill them with application specific codes.

Note that when the students were asked to develop the applications, they were provided with the required framework to read and write back an image file. They were also provided with the functions to convolute an image. The required algorithms (1) to divide the original image among the parallel processes, (2) to convolute each part of the divided image and (3) to combine convolute image partitions into one final convoluted image are also discussed elaborately.

Moreover, the source of the sequential program to convolute an image was posted along with the problem statement. In the given code, an image was represented with an object, named *Image*. An *Image* object represents an image (or part of an image) by storing the height, width and pixel data. The object was equipped with different methods to access different properties of the image. In the given sequential code of the program, the convolution operation was performed on an *Image* object. The code was arranged in this way to increase the reusability of the given code and also to ensure that students were able to emphasize more on parallelizing the application rather than learning in depth technical details.

5.1.2 Results

We asked the students to provide us some usability information. We were interested about different metrics, we described in the last subsection. We were also interested in code size, as another reusability factor. In this subsection, we summarize the information provided by the students.

Students reported that their average learning time to have a grasp of MPI environment was 11.4 hours. On the contrary, the learning time for SuperPAS was 15.2 hours. It took around 19.4 hours of time, on the average, for the students to write the image convolution program using MPI. To develop the required skeletons, the student needed to spend on an average 15.3 hours of time whereas they spent approximately 9 hours to concretize the abstract skeletons and to write the code-complete application.

Students also reported that they reused 40% code from the sequential program when they developed the parallel program on MPI environment. This reusability measure remains same (i.e., 40%) when SuperPAS system was used to develop the program. The student also reported that given the parallel MPI source code, the reusability factor went up to 80 $\sim$ 90%.

5.1.3 Analysis

Beside the above results, the students were also asked to compare their experience of using MPI and SuperPAS. In the following list, we comprehend the analysis of the results (from the last subsection) and the comments from the students.

- The SuperPAS model for parallel programming is more complex than the MPI model.

- The time to learn the SuperPAS model and the SDL is higher than that's of MPI model and MPI library.

- Developing parallel applications are significantly easy, if the required abstract skeletons already exist in the repository.

- The SuperPAS system becomes more beneficial, if the application in hand is complex with several skeletons and with a skeleton hierarchy of more than two levels.

- The object-oriented interface (the *Msg* object) and skeleton specific primitives for communication is easier to use than using the message passing functions in MPI.

5.2 Performance

In this section, we present the performance of the programs developed using SuperPAS. We also compare between different execution times of applications written using SuperPAS and MPI.

5.2.1 Environment

We used a private cluster to run the test applications to measure the performance. The cluster consists of 3 computers. Each computer has a single Intel Pentium 4 processor of 2.4 MHz and 256 MB of volatile memory. Those computers are connected through a 100 Mbps network. All the three computers are running Red Hat Linux version 9.0 [46] and LAM-MPI version 7.1 [41].

For the test purpose, we used the image convolution application that convolutes 30 images of with and height of 1280 and 1083 pixels respectively. We also developed

Table 4: Performance

Performance Metrics	MPI (second)	SuperPAS (second)
Time to complete image convolution	244.251	245.657
Total execution time to convolute images	244.536	246.358
Time to complete image processing	507.591	509.581
Total execution time to process images	508.238	510.643

another parallel image processing application to find the contours of objects in a image. Again, we used 30 images of size 1773 × 2352 pixel as the test case. Each image consists of a map of an area showing buildings and roads.

5.2.2 Results and Summery

Table 4 shows the performance of the applications written in both MPI and SuperPAS. To measure each of the metrics, we considered an average of 10 runs. The first row shows the time to execute the actual parallel image convolution algorithm. It does not include the MPI or SuperPAS environment setup time whereas the second row considers the environment setup time. The third row elaborates the time to run the actual image processing algorithm without considering the environment setup time whereas the times shown in fourth row includes the setup time. The setup time in MPI or SuperPAS includes the time of creation of the process hierarchy and communication channels (*communicators*, in MPI terminology).

From the second and forth rows, it may seem that the performance of SuperPAS applications are slightly worse than the MPI applications. In fact, the environment initialization step for a SuperPAS application is more complex than that's of the similar MPI application. However, it should be noted that this initialization takes place only once in the life time of the application. Though the initialization time grows with the complexity of the skeleton hierarchy, it will not increase with the increment of the life time of the application.

From the times listed on the first and third rows, it can be found that SuperPAS applications require around 0.411% and 0.392% more time respectively to have the solutions of the given problems. This slowdown indicates approximately 6 minutes of additional execution time for the applications running for a day. As the SuperPAS run-time system consists of a very thin layer over MPI, this performance degradation is expected. However, after contemplating the amount of slowdowns and the flexibility of development of the applications, we conclude that applications developed using SuperPAS do not show any noticeable performance degradation.

Chapter 6

Conclusion

Our research team is working towards making the PAS system a usable parallel programming tool with the added benefit of reusability as compared to MPI. SuperPAS is the first step towards that goal. SuperPAS is targeted to make PAS extensible and more flexible. In this thesis, we propose a model for designing abstract skeletons for the PAS system. This facility introduces a new category of users to the PAS system: the skeleton designers.

The skeleton designers are provided with a set of virtual processor grids and a rich set of communication and synchronization primitives. The designers specify the abstract topology of a pattern on top of those virtual processors. The pattern specific higher level communication and synchronization primitives are constructed using the given basic and lower-level primitives.

We also propose a language to implement the SuperPAS model. Using this language, users of the system can develop new skeletons according to the needs of the targeted parallel application. We also developed sufficient tools to realize the model in practice. According to the model, the underlying details are hidden from the users of the skeletons, i.e. application developers, so that they can concentrate more on the application development phase. The developed tools and run-time system are also ensured to hold this property. The run-time system is a complete object-oriented library. The system requires the users to fill in some specific methods and write their own objects, if required.

We tested some applications, developed using the SuperPAS system, from both the usability and the performance perspective. We have found that the SuperPAS eases the development process for big and complex applications. We have also found that in spite of the generality of the system, there exists no significant (less than 2%) performance degradation of the applications, developed using this system.

Though the current implementation of the SuperPAS system is sufficient, it is possible to add more features and tools to facilitate the users. Moreover, the model and the implementation is yet to support the linkage paradigm of aliasing. Visual tools would obviously further relax the development procedure. The system also lacs the support of a debugger which is capable of providing a view, conforming to the SuperPAS model, during the debugging time.

Besides SuperPAS, our research team is now working on several other issues of PAS system. As discussed in the introduction, till now we do not have any well designed performance modeling methods for parallel programs. Unlike parallel programs represented as generic graph, skeletons in parallel programs have some fixed structure and some fixed communication / synchronization policies. Our research team is investigating the scope for *performance modeling* of these regularities in PAS system.

The PAS model defines only the architectural aspect of a pattern inside a skeleton. However, to measure the run-time cost of a parallel program, the behavior of the pattern should be known. In the PAS system, skeletons handle the structural issues whereas the developers define the behavior. *Synchronous slicing* is the method to extract out the communication synchronization behavior of a given program. With the help of the synchronous slicing and performance model, we expect to be able to calculate the cost of a given parallel program.

Our team is also working on the issue of *static and dynamic optimization* of the PAS system. Those issues are important when we intent to run an already developed application on a newly bought parallel computer or when we start to design a parallel application for some specific target machine. The research goals are to find the model to optimize a given PAS-based parallel program for a parallel computer and also to find the model to develop a optimized parallel program for a particular parallel computer.

Hopefully PAS system will become an interesting and most used parallel programming model and environment in the near future.

Bibliography

[1] Top500. (2004) Top 500 supercomputer site. [Online]. Available: http://www.top500.org/

[2] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, USA, May 1993, pp. 1–12.

[3] J. Narem, "An informal operational semantics of C-Linda V2.3.5," Department of Computer Science, Yale University, CT, USA, Tech. Rep. 839, Dec. 1990.

[4] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. S. Jr, and M. E. Zosel, *The High Performance Fortran Handbook.* MIT Press, 1994.

[5] K. M. Chandy and C. Kesselman, "CC++: a declarative concurrent object-oriented programming notation," *Research directions in concurrent object-oriented programming*, pp. 281–313, 1993.

[6] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 4–14, Apr. 1994.

[7] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum, "Experience with distributed programming in Orca," in *IEEE CS Int. Conf. on Computer Languages*, New Orleans, Louisiana, Mar. 1990, pp. 79–89.

[8] K. Clark and S. Gregory, "PARLOG: parallel programming in logic," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, pp. 1–49, 1986.

[9] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, "The case for high level parallel programming in ZPL," *IEEE Computational Science and Engineering*, vol. 5, no. 3, pp. 76–86, Sept. 1998.

[10] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction.* New York, USA: Oxford University Press, 1977.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software.* New York, USA: Addision-Wesley Publishing Company, 1994.

[12] D. C. Schmidt, "Ace: an object-oriented framework for developing distributed applications," in *In Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, 1994.

[13] J. C. Browne, M. Azam, and S. Sobek, "Code: A unified approach to parallel programming," *IEEE Software*, vol. 6, no. 4, pp. 10–18, 1989.

[14] A. Singh, J. Schaeffer, and M. Green, "A template-based tool for building applications in a multicomputer network environment," in *Parallel Computing 89*, North-Holland, Amsterdam, 1989, pp. 461–466.

[15] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons, "The enterprise model for developing distributed applications," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol. 1, no. 3, pp. 85–96, 1993.

[16] A. Bartoli, P. Corsini, G. Dini, and C. A. Prete, "Graphical design of distributed applications through reusable components," *IEEE Parallel and Distributed Technology*, vol. 3, no. 1, pp. 37–50, 1995.

[17] S. Siu and A. Singh, "Design patterns for parallel computing using a network of processors," in *6th International Symposium on High Performance Distributed Computing (HPDC '97)*, Portland, OR, Aug. 1997, pp. 293–304.

[18] S. MacDonald, D. Szafron, J. Schaffer, and S. Bromling, "From patterns to frameworks to parallel programs," *Parallel Computing*, vol. 28, no. 12, pp. 1663–1683, 2002.

[19] D. Goswami, A. Singh, and B. R. Preiss, "From design patterns to parallel architectural skeletons," *Journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 669–695, 2002.

[20] M. Vanneschi, "The programming model of assist, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, 2002.

[21] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation.* Cambridge, Massachusetts: MIT Press, 1989.

[22] J. Darlington, A. J. Field, and P. G. Harrison, "Parallel programming using skeleton functions," in *Lecture Notes in Computer Science*, vol. 694, Munich, Germany, June 1993, pp. 146–160.

[23] I. Foster and R. Stevens, "Parallel programming with skeletons," in *ICPP'90*, 1990.

[24] D. Goswami, "Parallel architectural skeletons: Re-usable building blocks for parallel applications," Ph.D. dissertation, University of Waterloo, Canada, Oct. 2001.

[25] M. Forum. (2004) Message passing interface forum. [Online]. Available: http://www.mpi-forum.org/

[26] PVM. (2004) Parallel virtual machine. [Online]. Available: http://www.csm.ornl.gov/pvm/

[27] M. M. Akon, D. Goswami, and H. F. Li, "A parallel architectural skeleton model supporting extensibility and skeleton composition," in *Second International Symposium on Parallel and Distributed Processing and Applications (to be published by LNCS)*, Hong Kong, 2004.

[28] A. Singh, J. Schaeffer, and M. Green, "A template based approach to generation of distributed application using a network of workstations," *IEEE Transaction of Parallel and Distributed Systems*, vol. 2, no. 1, pp. 52–67, 1991.

[29] F. Chan, J. Cao, and Y. Sun, "High-level abstractions for message passing parallel programming," *Parallel Computing*, vol. 29, pp. 1589–1621, 2003.

[30] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* CA, USA: Morgan Kaufmann, 1992.

[31] M. J. Quinn, *Parallel computing: Theory and Practice.* New York, NY, USA: McGraw-Hill, Inc, 1993.

[32] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing.* Addison Wesley, 2003.

[33] F. P. Preparata and J. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation," *Communications of the ACM*, vol. 24, no. 5, pp. 300–309, 1981.

[34] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan, "Pattern-based parallel programming," in *2002 International Conference on Parallel Programming (ICPP-02)*, Vancouver, British Columbia, Aug. 2002.

[35] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, and S. MacDonald, "Using generative design patterns to generate parallel code for a distributed memory environment," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, California, 2003.

[36] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, P. Pesciullesi, R. Perego, and M. Vanneschi, "Assistconf: a grid configuration tool for the assist parallel programming environment," in *Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Genova, Italy, Feb. 2003, pp. 193–200.

[37] T. G. Project. (2004) Grid computing info centre. [Online]. Available: http://www.gridcomputing.com/

[38] R. Feldmann and W. Unger, "The cube-connected cycles network is a subgraph of the butterfly network," *Parallel Processing Letters*, vol. 2, no. 1, pp. 13–19, 1992.

[39] A. M. Despain and D. A. Patterson, "X-tree: A tree structured multi-processor computer architecture," in *Proceedings of the 5th annual symposium on Computer architecture.* ACM Press, 1978, pp. 144–151.

[40] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree: An index structure for high-dimensional data," in *Proceedings of the 22nd International Conference on Very Large Databases*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds. San Francisco, U.S.A.: Morgan Kaufmann Publishers, 1996, pp. 28–39.

[41] LAM-MPI. (2004) Local area multicomputing. [Online]. Available: http://www.lam-mpi.org/

[42] P. Wainwright, *Professional Perl Programming.* Birmingham, UK: Wrox, 2001.

[43] GNU. (2004) GNU's Not UNIX. [Online]. Available: http://www.gnu.org/

[44] H. R. Myler and A. R. Weeks, *The Pocket Handbook of Image Processing Algorithms In C.* Englewood Cliffs, N.J: Prentice-Hall, 1993.

[45] Department of Computer Science of Concordia University. (2004) Concordia beowulf cluster. [Online]. Available: http://www.cs.concordia.ca/Beowulf/

[46] Red Hat, Inc. (2004) Red hat linux. [Online]. Available: http://www.redhat.com/