Parallel Architectural Skeletons: Re-Usable Building Blocks for Parallel Applications

by

Dhrubajyoti Goswami

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2001

©Dhrubajyoti Goswami 2001

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-65241-6

Canada

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

In the context of object-oriented software design, the concept of design patterns is well studied and frequently applied. Similar ideas are being explored in other areas of computing as well. Over the past several years, researchers have been experimenting with the feasibility of employing design-pattern concepts in the parallel computing domain. Starting with the late 80s, several pattern-based systems have been built and several parallel programming models based on patterns have been formulated. As an important distinction with object-oriented patterns, most researchers here aim to use patterns not only at the design level, but also at the implementation level.

Though the idea of design- and implementation-level parallel patterns hold significant promise, most of the current pattern-based approaches to parallel programming suffer severe limitations, some of which include: lack of flexibility, limited to zero extensibility, ad hoc pattern sets, and language-related limitations.

In contrast to the previous approaches, this research proposes a generic, pattern-based model for fast and reliable development of parallel applications. The model is generic because it can be described in a manner independent of patterns and applications. The model is based on the message-passing paradigm, which makes it particularly suited for a network of workstations and PCs. The term *parallel architectural skeleton* is used to represent the generic set of attributes associated with a pattern. An architectural skeleton contains the necessary ingredients for constructing application-specific virtual architectures. Together with the complementary communication-synchronization protocols, a user can develop applications on these architectures.

The generic nature of the Parallel Architectural Skeleton Model (PASM) en-

hances usability. In addition, the model combines the flexibility of a low-level MPI-like message-passing parallel programming environment together with the benefits of high-level parallel patterns. This approach provides the necessary flexibility to the user in application development. Hierarchical pattern composition is an inherent characteristic of the model, which in turn facilitates hierarchical refinement.

PASM is an ideal candidate for an object-oriented style of design and implementation. An object-oriented and library-based implementation of the model, using MPI as the underlying communication-synchronization library, is completed without necessitating any language extension. The object-oriented and library-based implementation, together with the generic model, facilitates extensibility. That is, new patterns can be added to the system by an experienced user without requiring modifications to the existing repertoire.

A thin implementation layer over the standard message-passing interface, MPI, has resulted in negligible performance degradation. Moreover, from the software engineering perspective, desired software qualities such as *separation of concerns* and *software reusability* are some of the basic features of the approach. Other software engineering related benefits emanate from the aforementioned unique features of PASM, i.e., genericness, inherent support for hierarchical design and development, low-level flexibility, and an extensible repertoire of parallel architectural skeletons.

Acknowledgements

I would like to thank my supervisor, Prof. Ajit Singh, and co-supervisor, Prof. Bruno R. Preiss, for their guidance, support, and encouragement throughout my research. I would also like to thank all members of the Parallel and Distributed Systems (PADS) group at the University of Waterloo. The PADS meetings were extremely beneficial to convey my ideas, to improve my public speaking skills, and to listen to other's ideas.

I would like to acknowledge Mauricio De Simone and Stephen Siu for their initial contributions to this research. My special acknowledgment goes to Ladan Tahvildari for her active involvement in the software-quality experiments and analyses using the PASM system.

I will also like to thank all my friends and colleagues (specially Amir Gourgy and Mohammad Zulkernine from the Bell Canada Software Reliability lab, Rodrigo Fuentes-Loyola, Diego Hernandez, Nader Fayyaz, and the others at the Minota Hagey Residence) for their continual support, thought-provoking discussions, and friendship.

My special thanks go to my parents and my sister for their moral support throughout the research. Finally, I am grateful to God for providing me with the opportunity to conduct this exciting research, guiding me throughout, and for enabling me to conquer the mountain.

Contents

1 Introduction **1**

1.1 Objectives of this Research 4

1.2 Different Approaches to Parallel Programming 5

1.2.1 Advantages and Limitations 7

1.3 An Alternative Approach to Parallel Programming 8

1.3.1 Shortcomings of the existing pattern-based approaches . . . 10

1.3.2 Contributions of this research 10

1.4 Organization of the Thesis 12

2 Patterns in Parallel Computing **13**

2.1 Example 1: Divide and Conquer 16

2.2 Example 2: A Graphic Animation System 19

2.3 Example 3: Algorithmic Patterns 21

2.4 Existing Approaches to Parallel Programming 24

2.5 Motivation for Pattern-based Approaches 27

2.6 Some Existing Pattern-Based Approaches 29
2.6.1 Code 30
2.6.2 Frameworks 31
2.6.3 Enterprise 32
2.6.4 Hence 33
2.6.5 Tracs 33
2.6.6 DPnDP 35
2.6.7 Archetypes 37
2.6.8 Model Programming 37
2.6.9 Algorithmic Skeletons 38
2.7 Limitations of the Existing Pattern-Based Approaches 40
2.8 A Generic Model for Pattern-Based Parallel Computing 41

3 Parallel Architectural Skeletons 43

3.1 The Model 43
3.1.1 A formal description of the model 45
3.2 Examples 49
3.2.1 A Graphics Animation Application 50
3.2.2 Jacobi 54
3.2.3 Divide and Conquer 55
3.3 Summary 57

4 An Object-Oriented Implementation **59**

4.1 Basic Implementation Features 59

4.2 The Textual User Interface: Examples 61

4.2.1 Hello world 61

4.2.2 The graphics animation application 62

4.2.3 Jacobi 73

4.2.4 Divide and Conquer 75

4.3 Implementation Issues 78

4.3.1 Implementing Architectural Skeletons: Reusability and Extensibility 79

4.3.2 The Graphics Animation Application: Revisited 81

4.3.3 The Dynamic Execution Model 83

4.3.4 Mechanisms for Constructing the HTree 85

4.3.5 Obtaining Information about Peers 87

4.3.6 Process-Processor Mapping 89

4.4 Steps Involved in Building an Application 89

4.5 Summary 91

5 A Pattern Language **92**

5.1 Introduction 92

5.2 Pattern: Dynamic Replication 95

5.3 Pattern: Parallel Divide and Conquer 101

5.4 Pattern: Data-parallel computation 108
5.5 Pattern: Hierarchical Composition 109
5.6 Pattern: Pipeline 110
5.7 Pattern: Single process computation 111
5.8 Conclusion 111

6 Performance Evaluation **113**

6.1 Application Specific Evaluation 114
6.1.1 PQSRS 114
6.1.2 2-D Discrete Convolution 116
6.1.3 Jacobi 116
6.1.4 Software Quality Measurement 119
6.2 Application Independent Evaluation 119
6.2.1 Comparison of Some Basic Primitives with MPI 120
6.2.2 Effect of Granularity on Master-Worker Performance 122
6.2.3 Pipeline with and without Replication 123
6.2.4 Performance of Pipeline with Varying Granularity 125
6.2.5 Conclusion 127

7 Crucial Issues, Future Directions **128**

7.1 Fundamental Contributions 128
7.2 Software Engineering Issues 129

7.2.1 Reuse 129
7.2.2 Genericness 130
7.2.3 Flexibility 130
7.2.4 Extensibility 131
7.2.5 Hierarchical Development and Refinement 132
7.2.6 Separation of Concerns 133
7.2.7 Composition Using Patterns 133
7.3 Comparison with Related Work 134
7.4 Future Research Directions 137
7.5 Conclusion 138

Bibliography **140**

List of Figures

2.1 A divide-and-conquer tree 18

2.2 The graphics animation application 20

3.1 Application development using architectural skeletons 44

3.2 Structure of an abstract module 45

3.3 Diagrammatic representation of a HTree 48

3.4 Structure of the animation application 51

3.5 HTree representation of the animation application 52

3.6 Structure of the Jacobi application 55

3.7 Structure of a stand-alone divide and conquer application 57

4.1 Hello World 62

4.2 Structure of the animation application 63

4.3 Structure of the Jacobi application 73

4.4 High level class diagram behind the design of the skeleton library . 79

4.5 High level class diagram for the graphics animation application . . . 81

4.6 High level class diagram after refinement 82

4.7 HTree and its traversal scheme 83

5.1 Relationship between skeletons and patterns in the language 94

6.1 Speed-up ratio versus number of processors 115

6.2 Effect of granularity on speed-up 118

6.3 Effect of granularity on performance 122

6.4 Performances of pipeline with and without replication 124

6.5 Performances of pipeline with varying degrees of granularity 126

Chapter 1

Introduction

Parallel application design and development is a major area of focus in the domain of high performance scientific and industrial computing. In fact, starting from computational physics to weather prediction and space applications, parallel computing is becoming an integral part in several major application domains.

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 reasonable price-performance ratio. Contrast this to the situation a few years back, where parallel computing was confined only to special-purpose parallel computers, each priced high enough to be affordable only by major research/academic institutions. Consequently, high-speed networks and fast general-purpose computers are facilitating the mainstream adoption of parallel computing.

However, it must be emphasized that parallel computing is complex. Complexity of parallel software development has always been one of the major obstacles to the mainstream adoption of parallel computing. Though parallel computers, and lately

multiprocessor workstations, PCs and their clusters, are becoming more and more economical and widely available, their efficient utilization has been an issue of concern since the dawn of parallel computing.

There are several reasons for the aforementioned complexity 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 languages, compilers, compilation techniques and a group of programmers proficient in their use.

  Often a parallel algorithm is suitable only for specific types of programming models. As a result, algorithms developed for one platform are not always easily portable to other platforms, or may not be as efficient on other platforms if ported.

- In the case of parallel programming, there are added complexities over sequential code due to many of the low-level parallelism related details. These include problem decomposition (the identification of parallelism), distribution (the physical exploitation of the potential parallelism identified by decomposition), process/thread creation and management, process-processor mapping, communication and synchronization, data packing and unpacking, load balancing, and architecture- and network-specific low-level details. As a result of these low-level complexities, parallel programming remains an expert's job.

- In the case of sequential computing, it is possible to predict performance of an application (e.g., faster processor and memory access result in better performance). This predictability is more difficult with parallel computing. There exists certain parallel programming models (e.g., implicit data-parallelism as with functional languages) which hinder performance prediction on the part of the programmer. The actual performance depends on the efficient mapping of the parallel components over the multiprocessor architecture, and might depend on the order of evaluation of the implicit parallel components. Unless the programmer has explicit control over these issues, it is very difficult to correctly predict performance. On the contrary, explicit control over these issues results in another set of complexities as mentioned before. Consequently, a suitable compromise regarding how much of the low-level details are handled by the programmer is often needed.

- Due to the lack of a standard model of parallel computation, there are no standard methods or tools for developing, debugging or profiling of parallel applications. Consequently, specialized tools for individual platforms are often required.

- It is often impossible to reuse existing sequential application code while developing parallel applications. This lack of reusability results in writing everything from scratch. Also, porting an existing parallel application to another platform often requires major modifications to the existing parallel application.

- Another important issue, which applies to sequential computing as well, is the programming language. Researchers have been experimenting with different parallel programming languages and paradigms. Lack of a uniform language

across sequential and parallel computing platforms results in limited reusability of existing code. Moreover, frequent shifting to new languages or language extensions and new programming paradigms is problematic, because it often results in higher learning curves and non-reusable code. It is generally the case that a group of programmers comfortable with a specific model and a set of associated languages and tools find it difficult to shift focus to something entirely different.

Over the years, there have been numerous efforts to overcome some of the aforementioned difficulties. This research focuses on a specific approach to network-oriented parallel programming that is based on frequently used parallel design patterns.

1.1 Objectives of this Research

Starting with the early days of parallel computing, different abstractions and techniques have been proposed to handle some of the aforementioned complexities. In this research, we investigate one specific approach to parallel programming that is based on the use of frequently occurring structures for parallelism. These frequently occurring structures are often called *parallel design patterns*. Examples of such recurring patterns are: static and dynamic replication, divide and conquer, data parallel pattern with various topologies, compositional framework for irregularly-structured control-parallel computation, systolic array, pipeline, singleton pattern for single-process single- or multi-threaded computation.

In particular, it is believed that the high-level abstractions provided by such structures can be used to simplify the task of building parallel applications and to

promote software reusability, together with other issues which are discussed shortly. There have been other pattern-based approaches in the past. However most of them have severe limitations. This research continues the effort to overcome some of those limitations and to create a more flexible and usable pattern-based parallel programming environment.

In the following section, some of the different approaches to parallel programming are briefly discussed. These will be further elaborated in the next chapter.

1.2 Different Approaches to Parallel Programming

There are two main approaches to address the complexities of parallel computing mentioned earlier. The first approach is an architectural route that develops a *parallel Von-Neumann machine*, i.e., a universal abstract machine, to which a conventional parallel programming model can be applied with predictable performance, and that can be implemented on a scalable architecture with a predictable performance cost. For instance, the parallel random-access machine (PRAM) model [71] and the distributed shared memory model [45] take this route.

The second approach is based on high-level parallel programming models that attempt to hide the low-level details related to hardware architectures, interconnection topologies, process and thread creation/mapping, communication and synchronization, load balancing, data marshaling and un-marshaling, and numerous other details. Some of them also try to handle the issue of portability across various architectural platforms.

Different models employ different abstraction techniques such as communication

libraries, macros, new parallel languages and language extensions, and abstract data types. Depending on the amount of direct specification of parallel interactions required from a programmer, these models can broadly be categorized as explicit, implicit or semi-explicit.

In the explicit models, a user has to explicitly handle all the parallelism-related issues in the software. These approaches can be further classified based on their levels of abstraction. At the lowest level, the user works with primitives such as TCP/IP sockets [44] at a level closest to the hardware. Parallel programming using sockets is probably as difficult as sequential programming using assembly language. As a result, higher level parallel programming models and tools have been developed on top of sockets [27, 28, 35, 72]. This process is similar to employing high level programming languages to hide the difficulties associated with assembly language programming.

At the other extreme of the existing parallel programming models, there is no explicit specification of parallelism in the user-supplied application code. Here, the user writes sequential code and the parallel programming system, for instance a parallelizing compiler, explores the parallelism in the code. Parallel code is then automatically generated by the compiler for the underlying architecture [5]. Various functional programming languages [29] are implicitly parallel, which can be exploited by the associated compilers.

There are other approaches which fall in between the above two extremes and, therefore, are referred to as semi-explicit parallel programming models. In some of these approaches, the user handles the performance-crucial issues (for instance, data and task decomposition, resource allocation) and the rest is handled by the system [21, 46, 56, 63].

These approaches are further discussed in the next chapter.

1.2.1 Advantages and Limitations

All high-level approaches to parallel programming have the obvious advantages of reducing some of the difficulties associated with parallel programming. However, many of them have their own limitations, which often overshadow the gains achieved.

It has been found that the PRAM model imposes unavoidable overhead when implemented on certain architectures (e.g., the distributed memory machines, which are the most scalable), and hence efficient implementation on these architectures is impossible.

The parallel programming models, in general, have at least two main objectives: to provide high-level abstractions to free the user from the low-level details, and to retain the good performance and flexibility generally available with low-level primitives. However, gains in some of these objectives often involve trade-offs in some other issues. For instance, it is possible to obtain near optimal performance with hand-crafted parallel code written close to the hardware level. However, as the abstraction level increases and the more one relies on automatic parallelization, the performance gradually degrades unless it is possible to perform some user-level fine-tuning of the generated code.

Many of the high-level parallel programming approaches are suitable only for solving a limited range of problems that fit into specific models of parallel computation. For instance, the success of parallelizing compilers is limited by the availability of parallelizable loops inside an application. Similar situations apply to the functional programming languages, many of which are implicitly data-parallel.

Most of these languages exhibit implicit parallelism, and consequently all crucial parallelism-related details need to be handled by the language implementer. However, somewhat successful implementations of these languages have been achieved only on shared-memory architectures.

With high-level models, the higher-level of abstraction is often associated with a decline in flexibility on the part of the user in application development. This means that the user is often restricted to working with high-level abstractions, without being able to use lower-level primitives like point-to-point message passing for customization or better performance. Thus, in most cases, it is impossible for an experienced user to fine-tune an application or to extend the existing system as need arises (i.e., lack of extensibility). In fact, the loss of flexibility is a major issue that dictates other extremely important issues like usability.

Finally, the lack of portability of high-level parallel systems across various architectural platforms and the non-availability of support tools remain chronic problems in the area of parallel programming. These and several other issues are further elaborated in the next chapter.

1.3 An Alternative Approach to Parallel Programming

This research focuses on an alternative approach to parallel programming that is based on the idea of parallel design patterns. In the context of object-oriented software design, the term *design pattern* is used to describe strategies for solving recurring design problems in systematic and general ways [25]. In a similar fashion, *parallel design patterns* specify recurring problems in the parallel computing domain

and their solution strategies.

It has been observed that a large number of parallel applications are based on commonly occurring control structures and that they differ only in the application specific code [9, 10] and some other application specific parameters. As a result, it is often possible to achieve a significant amount of *separation of specifications*, whereby these parallel structures can be generated independent of the application code. The application code can later be plugged into the generated parallel structures. An isolated parallel structure or a composition of them may constitute the skeleton of a parallel application, i.e., it embodies the parallel structure of the entire application without the application specific code. Since the generated skeleton hides most of the lower-level details, the developer is freed from the extra burden of these low-level complexities. A design pattern based parallel programming system helps the user to generate these parallel structures. A detailed discussion of this approach can be found in the subsequent chapters.

Starting with the late 80s, several pattern based approaches were studied and some systems developed. All these approaches use patterns as application independent reusable building blocks that hide most of the low-level and error-prone parallelism related details. An important distinction in the use of pattern-concepts in parallel computing is that researchers here use patterns not only at the design level but also at the implementation level. Accordingly, in the rest of the thesis, the terminology *parallel pattern* is used to imply both design- and implementation-level patterns.

1.3.1 Shortcomings of the existing pattern-based approaches

Though the idea of design- and implementation-level patterns holds significant promise, in practice however, most of the existing pattern-based systems face some or all of the limitations mentioned previously, e.g., lack of flexibility, limited to zero extensibility. Moreover, most of these systems support only a limited and fixed set of patterns in ad hoc manners, which often results in confusion regarding their use. Due to the ad hoc nature of their pattern components, pattern composition is often impossible inside these systems (or possible in very restricted manners), thus further limiting their use. The ad hoc approach also leads to limited extensibility. Many of these systems are based on new languages or language extensions, thus contributing to another bottleneck. Each of these factors limits the usability of a particular system.

Detailed discussions on some of these approaches and their shortcomings are postponed till the next chapter.

1.3.2 Contributions of this research

As opposed to the previous ad hoc approaches, this research proposes a generic pattern-based model for fast and reliable development of parallel applications. The model is generic because it can be described in a manner independent of patterns and applications. The model is based on the message-passing paradigm, which makes it particularly suited for a network of workstations and PCs. All of the parallel patterns mentioned previously can be elegantly realized within the frameworks of the model. The term *parallel architectural skeleton* is used to represent the set of generic attributes associated with a pattern. An architectural skeleton contains the necessary ingredients for constructing application-specific virtual architecture(s).

Together with the necessary communication-synchronization protocols, a user develops applications on the virtual architectures.

The generic approach of the Parallel Architectural Skeleton Model (PASM) helps in more than one aspects. It helps a user to become familiar with the approach, which results from the inherent commonality among the multiple patterns. It helps in pattern composition, due to the inherent presence of standard interfaces and protocols. Finally, it helps in designing new patterns, and thus, further extend the system. All these issues enhance usability.

Both low-level message-passing (something similar to PVM [27, 28] and MPI [35, 72]) high-level patterns are encompassed within the framework of PASM. Support for the low-level primitives, in conjunction with the high-level patterns, substantially enhances a user's flexibility in application development (a user has the options of using existing patterns, or developing applications from scratch using the low-level functionalities, or designing new patterns to incorporate into the system if need arises).

PASM can be well represented through the object-oriented style of design and implementation. An object-oriented and library-based implementation of the model in C++, using MPI as the layer underneath, is complete without necessitating any language extension. The generic approach in conjunction with the object-oriented and library-based implementation facilitate extensibility, i.e., new patterns can be added to the system by an experienced user without requiring any major modifications to the existing repertoire.

Hierarchical pattern composition is an integral characteristic of the model that facilitates hierarchical refinement. A pattern can be stand-alone or it can be contained inside another pattern. This capability enables multiple patterns to work

together based on a generic scheme. Support for hierarchical design and low- as well as high-level protocols provide added flexibility not found in existing parallel systems that aim to support design patterns.

Lastly, from the software engineering perspective, desired software qualities such as *separation of concerns* and *software reusability* are some of the basic features of PASM. Software engineering related aspects of the model are discussed in a later part of the thesis.

1.4 Organization of the Thesis

The next chapter further elaborates the different approaches to parallel computing and discusses some of the existing pattern-based approaches in some detail. Chapter 3 introduces the architectural skeleton model and illustrates the idea behind the model with several examples. Chapter 4 discusses an object-oriented implementation of the model and revisits the examples discussed in chapter 3. Chapter 5 revisits the model from the perspective of a pattern language and it also serves as a catalog of patterns. Chapter 6 discusses the various performance measures of the framework that implements the model. Finally, chapter 7 discusses several crucial software engineering related aspects of the model together with its comparisons with other related works and various other issues that need to be considered in the future evolutions of the work.

Chapter 2

Patterns in Parallel Computing

The term *design pattern* has been used by different researchers in different application-domains and at different levels of abstraction. For instance, one of the most prominent contexts in which design patterns are frequently applied is in the domain of object-oriented (abbreviated OO) software design. Here, design patterns imply recurring design problems in the OO paradigm and their solution strategies [25]. These OO patterns are not pre-implemented code in some particular language. Rather they document the methodologies for solving recurring design problems in systematic and general ways. The OO patterns need to be implemented (or, re-used from existing code with modifications based on the application's context) each time they are applied.

There are also patterns and pattern-based development toolkits in the domain of network-level distributed programming. For instance, ACE (the Adaptive Communication Environment) [57] is an OO toolkit that implements various network-level patterns to simplify the development of concurrent, event driven communication software. The design and implementation of ACE is based on fundamental com-

munication software design patterns [58]. The "Pattern Languages of Program Design" series of books [37] are good references covering pattern-related topics for a diverse range of disciplines.

In the parallel computing domain, design-pattern related concepts have been employed as early as in the late 80s. Different researchers have used different terminologies for describing similar concepts. However, different terminologies describe patterns at different abstraction levels, and they are often based on completely different methodologies. For instance, the term *design pattern* has been used to denote commonly occurring parallel or distributed computing abstractions [64]. Some other authors have used terms like *programming paradigm* [9], *algorithmic skeleton* [18] or *template* [62] to denote similar ideas. Different researchers have used patterns at different levels of abstraction. For instance, templates in [62] have completely different functionalities from the so called patterns in [64], which is evident in a later part of this chapter. Different approaches employ different methodologies for abstracting patterns. For instance, the algorithmic-skeleton stream of research treats patterns as algorithmic abstractions realizable as high-order functional constructs with associated cost functions. There are even variations inside the same research-stream. For instance, different authors have formulated their own versions of algorithmic skeletons, a comparison of which can be found in [14].

As an important distinction with the abstract-level OO design patterns in [25], most researchers in parallel computing have used patterns not only at the design level but also at the implementation level, i.e., the design-level patterns are also pre-implemented. This approach is similar in concept to a *framework* [40] from the Software Engineering perspective. The Software Engineering aspects of this research are discussed in the later part of the thesis.

In this thesis, the terminologies *parallel pattern* and *pattern* are used interchange-

ably to imply recurring design- and implementation-level patterns in parallel computing, unless otherwise specified (e.g., OO patterns to imply design patterns in the object-oriented context). Examples of such recurring patterns in parallel computing are: static and dynamic replication, divide and conquer, data parallel (with various topologies), pipeline, compositional framework for irregularly-structured control-parallel computation, systolic arrays, and singleton (for single-process, single- or multi-threaded computation).

As the examples in the previous paragraph suggest, a pattern in parallel computing is an application independent abstraction with associated structural and behavioral components. The same pattern applies to a wide range of different parallel applications. In other words, distinct parallel applications are found to possess identical structural and behavioral characteristics, and hence can be said to follow an identical pattern of parallel computation. One of the structural components might be the interconnecting topology of the various sequential-computing elements constituting the parallel-computing structure. At the same time, the behavioral components specify the unique behaviors associated with the structural components. For instance, a 2-D mesh for data-parallel computation and a systolic array might look identical from the structural perspective, but they have clearly distinct behaviors. Evidently, behavioral components play important roles in defining a pattern.

The presence of the same generic structural and behavioral components in a wide range of applications results in a number of beneficial aspects. *First*, each component can be studied in detail and its various properties can be recognized and documented for future use. Accordingly, the second time such a pattern is encountered, one does not have to start from scratch. *Second*, it is possible to abstract the application independent components associated with a pattern and implement

them as reusable modules for use in different applications. These reusable modules hide most of the low-level parallelism-related details (e.g., problem decomposition and distribution, process/thread creation, process-processor mapping, load balancing, communication and synchronization, data marshaling and un-marshaling, actual hardware architecture and topology) and thus enable the user to concentrate more on the application. Moreover, these pre-packaged modules are tested to be reliable, provided they are used correctly. *Third*, as already mentioned, the generic components are application-independent. Consequently, a clear *separation of specification* can be achieved, whereby it is possible to generate (and compile and run) the code-skeleton of an application, which is devoid of any application code. Such a clear separation not only liberates the user from the additional burden of the application-independent details, but also facilitates the reuse of sequential code segments of an application. As is illustrated later on, a parallel application can be viewed as a restructuring of the original sequential code with embedded parallelism constructs. With proper restructuring, it might be possible to reuse sizeable portions of the original sequential code.

The following examples further elaborate the concepts behind parallel patterns.

2.1 Example 1: Divide and Conquer

As a first example, let us consider the *divide and conquer* pattern. It is encountered in a large number of applications, starting from various sorting algorithms (e.g., merge and quick sort) to graph and matrix multiplication algorithms [54]. An algorithm that follows the divide and conquer pattern can be divided into two parts: (1) *Divide*: (recursively) divide the problem to be solved into smaller subproblems, except for the base case where the subproblem is directly solved by applying some

suitable algorithm without any further sub-division. (2) *Conquer*: the solution to the original problem is formed by combining the solutions to the subproblems.

As can be seen in the previous paragraph, it is possible to describe the generic divide and conquer pattern, without even considering a specific application. In other words, it is possible to abstract the high-level application-independent components of the pattern. It is now possible to dig further to sub-classify the high-level application-independent components. First let us identify one of the structural components: topology. It can be seen that repeated division of the problem into smaller sub-problems results in a tree-structured topology (refer to Figure 2.1). The original problem is input and output at the root of the tree. For each node of the tree during the divide phase, if the node is a leaf node (in other words, the base case) then the problem is solved applying some suitable algorithm; else the problem is further sub-divided and distributed to the children of the node. At each non-leaf node of the tree during the conquer phase, results from the children are combined to output the final result.

As one of the behavioral components, the tree can be static (independent of the base case and the problem size) in nature or it can be dynamic, i.e., grows in size top-down starting from the root of the tree during the divide phase, and shrink in size bottom-up towards the root of the tree during the conquer phase (Figure 2.1). Parallelism is obvious, i.e., each node of the tree can be an independent process or thread. However, it should be mentioned here that the type of parallelism achieved in divide and conquer is restrictive, and hence, not very efficient (e.g., during the conquer phase, each node in the tree has to wait before getting the results from all its children, which results in inefficiency).

Another behavioral component is the communication-synchronization pattern between the different nodes of the tree. The following high-level algorithm sum-

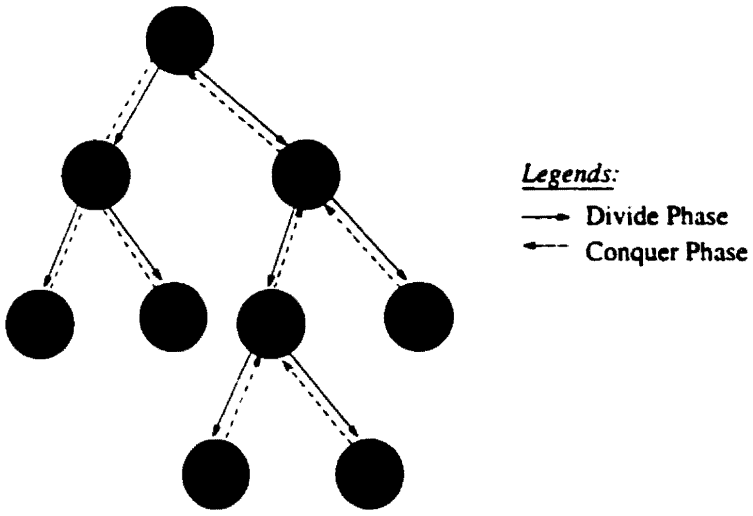


Figure 2.1: A divide-and-conquer tree

marizes the computation and communication involved at each node of a dynamic divide and conquer tree:

```
Input: data of size N.
Output: data of size M.

step 1. If the base condition is met, then process data sequentially and
        goto step 6, else
step 2. divide data into K subparts based on some criterion.
step 3. distribute the K subparts to K children.
step 4. collect results from the K children.
step 5. combine results to produce the final output.
step 6. output result.
```

In the preceding algorithm, communication is involved during the input and the output (step 6) phases of each non-root node. Moreover, there is communication involved with each non-leaf node during steps 3 and 4. The remaining steps involve sequential computation.

Until now, it has been possible to explain the parallel divide and conquer pattern without resorting to any specific application. As is shown in a later part of

the thesis. it is possible to abstract and implement the various structural and behavioral components associated with the divide and conquer pattern in a generic, application-independent manner, as re-usable module(s). The application-specific components can later be plugged into the generated structure. Where do the application-specific components fit? Various divide and conquer applications differ from one another in the data-types and data-structures used, and in the actions during step 1 (the base case and the sequential algorithm applied), step 2 (the data-division algorithm) and step 3 (the data-combining algorithm). Clearly, these are the places where the application-specific components fit-in.

The next example illustrates a case where more than one pattern is applied in a single application. The example uses two patterns, namely *pipeline* and *replication*, and also illustrates the concept of *refinement*, which is described in a later part of the thesis.

2.2 Example 2: A Graphic Animation System

This example demonstrates the use of pattern-based methodologies in the systematic development of parallel applications. Let us consider a graphics animation program consisting of three modules: *Generate*, *Geometry* and *Display* [63]. The program generates a sequence of graphical image-frames. Depending on the subject of animation, *Generate* computes the location and motion of each object for each frame. It then passes the frame to *Geometry*, which performs actions such as viewing transformations, projection and clipping. Finally, the frame is passed to *Display*, which performs hidden-surface removal and anti-aliasing, and finally saves the frame on the disk. The whole process repeats with successive frames, thus keeping the pipeline full during successive iterations.

The simplest way to parallelize this application is to use a 3-stage pipeline pattern and then to plug in the code for the three sequential modules into the three pipeline stages, as illustrated in Figure 2.2(a). Based on the particular method of implementation, modifications might be necessary in each sequential module to interact with the next stage. Other than that, the core of the sequential code remains intact. When the pipeline is full, each stage works on a different frame at an instant.

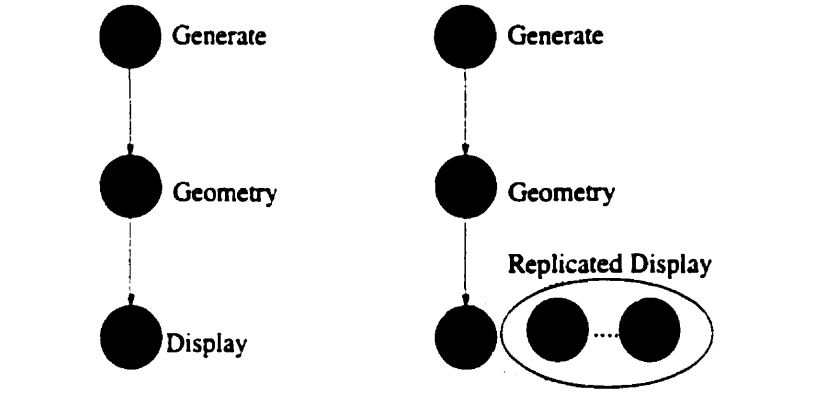


Figure 2.2: The graphics animation application

It is generally the case that the *Display* module, which performs actions such as hidden surface removal and anti-aliasing is the most time intensive of the three modules. This will slow down the entire pipeline. One way to resolve this bottleneck is to replace the stage 3 of the pipeline with a replication pattern, where as many copies of *Display* are dynamically created as needed (refer to Figure 2.2(b)). Consequently, each replicated *Display* module is working on a different frame, and this should speed up the entire application. It is interesting to note that the two applications remain the same in terms of functionality, but only differ in terms of

the patterns they use. The application codes for stage 1 and 2 remain unchanged, i.e., the modules *Generate* and *Geometry* remain untouched by this change. This type of localized replacement is called *refinement*. The example also illustrates the possible involvement of multiple patterns in a single application. This example and the idea of refinement are re-visited in a later part of the thesis.

Until now, patterns have been illustrated from the architectural perspective, i.e., patterns consisting of pure structural and behavioral components. Others have also investigated patterns from the algorithmic perfective (e.g., the *algorithmic skeletons* research, to be discussed later in this chapter). The next example illustrates this perspective.

2.3 Example 3: Algorithmic Patterns

Anyone familiar with any sequential programming language is knowingly or unknowingly dealing with many of the sequential computational patterns. For instance, let us consider the *while loop* in any of the imperative languages. In one sense, it is a computational pattern (may be used for both sequential and implicitly parallel loops), which can be described in an application independent manner as follows:

```
while <some_condition_is_true>
   <body_of_the_while_loop>
```

Irrespective of the application code, all while loops follow this particular pattern. In fact, any sequential programming language is made up of a collection of such useful patterns, which in effect determine the strength and the applicability of the language to a wide range of applications.

Can a similar technique be applied to develop a *universal parallel language* consisting of all useful parallel algorithmic patterns captured at the high-level language? This is a good proposition, however with no convincing solution till this moment. The diversity of parallel algorithms, models, and the underlying architectures are some of the factors to blame. However, the approach is worth mentioning.

One convenient way to express parallel algorithmic patterns is found to be through the use of high-order, polymorphic functional constructs. A high-order function is one which can take other functions as its parameters and can return another function. As an example, let us consider the polymorphic function *map*, which takes another function *f* as its parameter to return the function *map f*:

```
f:a -> b
(map f):[a] -> [b]
```

Here *f* is a function from *a* to *b*, where *a* and *b* can be any types, even identical. The result of applying *map* to *f* is the function *map f*, which takes a list of elements of type *a* as its argument and returns a list of elements of type *b*. The effect of *map f* is to apply *f* to every element of the argument-list and to place the result in the corresponding place in the result-list. The function *map* is polymorphic because f can be any suitable single-argument function. For instance, let us consider the following:

```
square:int -> int
(map square):[int] -> [int]
```

In the above, the function *map square* squares all the elements of an integer-list. The type of *map* can be mathematically expressed as follows, which states that *map* takes a single-argument function as its parameter and returns another function:

```
map:(a -> b) -> ([a] -> [b])
```

As can be seen, there is implicit data parallelism in the function *map f*, i.e., the same function *f* is being applied to different data-elements in the argument-list. In addition, the representation of the type of data-parallelism through *map* is completely architecture-independent, i.e., the previous representations have no reference to the underlying architecture (and applies to a sequential architecture as well).

Such types of suitable high-order polymorphic functions, which can conveniently express parallel computational patterns of some sort, are traditionally called *algorithmic skeletons*. Clearly, high-order polymorphic functions are most conveniently realized using the various functional programming languages. However, conventional imperative languages can also be used to represent higher-order functions, in which case they are implemented as *program-* or *procedural-templates* [18].

To conclude this discussion on examples, let us illustrate the already discussed divide-and-conquer pattern as a high-level functional construct, the same way it is represented in [18]. Here, *D_C* stands for the high-order polymorphic function representing divide-conquer, and *P* stands for the problem of type *prob*:

```
D_C indivisible split join f = F
                  where F P = f P, if indivisible P
                            = join (map F (split P)), otherwise

where,
    indivisible : prob -> bool
              f : prob -> sol
          split : prob -> [prob]
           join : [sol] -> sol
              F : prob -> sol
            map : (prob -> sol) -> ([prob] -> [sol])
            D_C : (prob -> bool) -> (prob -> [prob])) -> ([sol] -> sol)
                                 -> (prob -> sol) -> (prob -> sol)
```

All functions except *D_C* and *map* are application specific and need to be filled in by the user in some suitable base language, which need not be the same as the

high-level functional language in which the algorithmic skeleton is defined. This discussion on algorithmic skeletons is re-visited in a later part of the chapter. Now, before proceeding further, let us briefly discuss some of the existing approaches to parallel computing.

2.4 Existing Approaches to Parallel Programming

As is discussed in Chapter 1, the different models for parallel programming can be broadly divided into explicit, implicit and semi-explicit categories. Different models employ different levels of abstraction in an effort to hide the low-level details and, in order to achieve this, they use different abstraction techniques (e.g., communication libraries, macros, new parallel languages and language extensions, and abstract data types).

The different models falling into the explicit category can be further sub-classified based on their levels of abstraction. At the lowest level, which is closest to the hardware architecture, we have something similar to the TCP/IP sockets [44]. Sockets are the most difficult to use, but are the most flexible. Working with sockets is analogous to working at the assembly-language level. At a slightly higher level, we have the message passing libraries (MPL) and remote procedure call (RPC) [7] packages, which abstract lower level socket communication. Two standards have emerged for MPLs: the de facto standard PVM [27, 28] and the proposed message-passing interface (MPI) standard [35, 72]. Both standards are supported on many platforms. Most operating systems support some variants of RPC. Both MPL and RPC are quite popular among the parallel programming community. However, they are still a low level of abstraction and a programmer still has to tackle many lower-level details before correctly running his application. Consequently, other high-level sys-

tems have been built, including the system based on this research, on top of these packages with varying degrees of flexibility. These systems can be categorized from explicit to semi-explicit (e.g., Frameworks [61], Tracs [6], PUL-TUF Library [70]).

In the implicit models, parallelism related aspects are completely hidden from the user. One approach is through the use of parallelizing compilers, which explore the presence of implicit parallelism in sequential code. Different compilation techniques for high performance computing are discussed in detail in [5]. Ideally, given a sequential program, a parallelizing compiler is supposed to generate an efficient parallel program for the underlying architecture. However, the utility of currently available techniques for parallelizing compilers is limited by the fact that it can only deal with the parallelization of loops, which results in data-parallel applications. If an application involves control parallelism or if recursion, dynamic data structures or pointers are used, the present technology of parallelizing compilers is inadequate to automatically generate parallel programs from sequential code. Moreover, often the data partitioning generated by parallelizing compilers may not be as efficient as created by an expert programmer.

Another approach towards implicit parallelism is through the use of the different existing functional programming languages [29]. Such implicit parallelism is already illustrated in some of the examples in the previous section, e.g., *map* and *D_C* functions. However, in practice, the expressive power of these languages to represent the different existing parallel programming models is very limited. Besides, somewhat efficient implementations of some of these languages exist only for the shared-memory architectures [41, 67]. The theoretical work in [13] shows that no fully automatic scheduling strategy for functional languages on distributed-memory machines can ensure good performance, unless the processes can somehow migrate from one processor to another.

In the semi-explicit category, user takes care of the difficult parallelism related issues, the rest is handled by the system, for example: High Performance Fortran (HPF) [46] and Fortran D [38]. Both approaches follow the data-parallel model of computation. The user specifies the most crucial parts of parallel application development, e.g., the virtual architecture and the data decomposition strategy, and the rest is handled by the semi-automatic compiler and the run-time system. Like automatic parallelizing compilers, the utility of these approaches is limited by the availability of parallelizable loops resulting exclusively in data-parallel computations.

Several variations of the algorithmic-skeleton approach (refer to the previous section) are also semi-explicit. For instance, in the algorithmic-skeleton approach taken by Darlington et al [21], critical issues like resource allocation are documented for each skeleton/machine pair and are addressed explicitly during implementation in an interactive manner.

The Linda model [15] introduces the concept of shared tuple-space between processes, and based on individual perceptions it can fall anywhere from implicit to semi-explicit categories. Together with a few tuple-space operations, Linda is a powerful parallel programming model from the theoretical point of view. However, from the practical stand-point, implementation of the shared tuple-space in a distributed-memory architecture results in considerable communication overhead. Another weakness of Linda lies in the fact that it completely hides the cost of computation from the programmer, because nothing can be assumed about the response time of tuple-space accesses [67]. Implementation(s) of Linda over popular language platforms exist [43]. Irrespective of the weaknesses, the idea of tuple-space is an interesting one and is borrowed by some of the industrial tool-kits for building distributed applications [24].

There exist numerous other high-level parallel programming models and systems, which are not based on the data-parallel paradigm, and can be classified into the semi-explicit category. In these approaches, the user takes care of critical issues like data partitioning and mapping. The associated models provide functionalities to handle the rest of the complexities. Several of these approaches, which are based on C++ or its extensions, are discussed in [74]. Several pattern-based systems are also semi-explicit. Some of these systems are discussed in a following section.

2.5 Motivation for Pattern-based Approaches

Irrespective of whether it is communication libraries, macros, language extensions, parallelizing compilers, or application specific parallel libraries, all of these approaches intend to provide a higher level of abstraction to make the task of developing parallel applications easier by hiding the low-level details. However, all these efforts come with certain cost, and hence, there is always a trade-off involved. For instance, higher-level models make programming more restrictive (i.e., the programmer often loses flexibility in application development). Moreover, most high-level models are applicable only to a limited spectrum of parallel computing models (e.g., a good number of them are data-parallel).

Like any other high-level approach, the pattern-based approaches also have the same incentives and also fall into the trap of the similar trade-offs mentioned before. However, as compared to the other approaches, patterns have the potential for attaining the following additional objectives:

- **Re-usability**: Pattern-based approaches favor two types of reuse: reuse of application code and reuse of code for patterns (i.e., code-skeletons). One

main intention of all pattern-based approaches is to be able to reuse existing sequential code, instead of re-writing parallel applications from scratch. Patterns are application-independent abstractions. It is possible to generate code-skeletons for patterns and then plug in application-specific code inside these skeletons. This clear separation (also known as *separation of specifications*) might enable large segments of existing sequential code to be reused, because most parallelism-related constructs are abstracted inside the code-skeletons. Moreover, each pattern is itself a reusable component, realized as a pre-implemented code-skeleton, which can be reused in different applications that follow the same pattern.

- **Problem decomposition and distribution**: Problem *decomposition* means identifying the parallel components in a problem, while *distribution* deals with the suitable distribution of these parallel components to the underlying architecture. Through the use of patterns, the parallel components are already identified, e.g., in the divide-and-conquer tree, each node of the tree is a parallel component. Accordingly, a suitable architecture-specific distribution strategy for these parallel components, in order to minimize communication-synchronization overhead, can be laid-out by the implementer in an application-independent manner. Thus, by selecting a particular pattern, the programmer has selected a specific problem-decomposition and distribution strategy [18].

- **Usability**: Design patterns might simplify complex problems by letting developers approach them at a higher level of abstraction. Together with a thorough study and documentation of each pattern, it might be possible to reduce the time to understand a pattern and to use it properly. In fact, what

this research demonstrates is the feasibility of a well-defined, hierarchical-development model for systematic development of parallel applications.

- **Correctness:** The reusable code-skeletons for patterns can be well tested. This reduces the probability of erroneous code, provided patterns are used correctly. Most parallelism related and error-prone issues are hidden inside the code-skeletons. Consequently, the developer can spend more time in the application-specific issues.

The next section presents an overview of some of the existing pattern-based systems and similar approaches.

2.6 Some Existing Pattern-Based Approaches

The pattern-based approach to parallel programming is not new. It was applied in the late 1980s in systems like CODE [11, 12] and FrameWorks [60–63]. Some recent systems based on similar ideas are CODE2 [12], Enterprise [56, 63], HeNCE [12], PUL-TUF [70], Tracs [6], DPnDP [64-66] and CO_2P_3S [47]. In [20], the authors take a similar approach from the functional programming viewpoint (based on the functional language called FP, first introduced by Backus in late 1970 [4]). Similarly, in [22], the authors take their approach based on a high-level parallel programming language called Strand [23], which is similar to any logic programming language that uses guarded rules. Both model programming [9, 10] and Archetypes [16] emphasize the use of patterns from the viewpoint of education, documentation and example implementations. In the work by Pandey et al [53], the authors propose a concurrent programming model and a programming language (e.g., CYES-C++),

which emphasize the idea of separating computation from communication and synchronization that could facilitate extension and modification of programs (similar to the idea of *separation of specifications*, mentioned previously).

Some of these approaches are briefly discussed next from the point of view of their design philosophy, and definition and implementation of pattern-concepts. The systems surveyed here often have other contributions to parallel computing, other than the use of patterns. However, they are analyzed from the perspective of the use of pattern-related concepts in their programming models.

2.6.1 Code

CODE(Computationally Oriented Display Environment) [11, 12] was developed at the University of Texas at Austin in the late 80s. It is one of the pioneers of the idea of *separation of specifications*, by allowing a two-step development process. During the first step, programmers design the various sequential components and then, in the second step, compose them into a parallel structure. It uses visual programming techniques to aid the programmer graphically develop a parallel structure through the use of nodes and arcs that represent computations and interactions respectively. The programmer subsequently configures the nodes and arcs using textual annotations, following specific rules, in the graph. The sequential code can be developed in C or Fortran.

CODE follows a data-flow model of computation, where each node can begin computing only when data is available on each of the arcs incident to it. There is, thus, one obvious pattern in CODE, which is a composition of the various nodes and arcs interacting in a data-flow manner. Each node in CODE can itself be another data-flow graph. Thus, it supports reuse of other data-flow graphs by allowing

recursive embedding of graphs.

2.6.2 Frameworks

Frameworks [60–62] was developed at the University of Alberta in the late 80s. It was specifically designed to restructure existing sequential programs to exploit parallelism on workstation clusters. Accordingly, one of its main emphases was on the reuse of the existing sequential code. 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, which are at a different level of abstraction than the parallel patterns mentioned previously in this chapter. 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). These calls can be blocking or non-blocking. Messages between modules are in the form of user defined frames, which are C structures except that pointer types are not allowed. 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. A module's complete interconnection with other modules are specified by an input template, an output template, and a body template. An input template specifies the scheduling algorithm for incoming RPCs, an output template specifies scheduling algorithm for outgoing RPCs, and the body template specifies how the body behaves, either as a single node or as a replication. 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.

Frameworks is an early system that successfully exploited the idea of using commonly occurring parallel structures in parallel application development and inspired the development of another pattern-based system called Enterprise, discussed next.

2.6.3 Enterprise

Enterprise [56, 63] was developed at the University of Alberta and is 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 design, 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. This separation of specifications is much stricter in Enterprise than in Frameworks. Assets can be hierarchically combined to form a parallel program. Enterprise allows the use of pointers as parameters and the system takes care of marshaling and un-marshaling of data. Features like *future* variables enable more concurrency. A number of other features and tools improve the usability and portability aspects of

the system.

Like FrameWorks, Enterprise provides a fixed number of hard-coded patterns for application development. The patterns are built into the system and work well in combination with each other. However, there is no easy way of introducing new patterns without performing a major modification to the system. Besides, both Frameworks and Enterprise offer their own high-level models for application development, which users often found too restrictive and inflexible [63].

2.6.4 Hence

Hence (Heterogeneous Network Computing Environment) was developed at the University of Tennessee [12]. It is similar in purpose to Code, and uses similar visual programming and separation of specification techniques. However, unlike the data-flow graphs in Code, graphs in Hence depict control-flow. Hence supports patterns supporting replication, pipeline, loop and conditional constructs. It uses PVM underneath and runs on a network of Unix machines. The sequential procedures in each node are written in C or Fortran.

It was observed that Hence is much easier to learn and use as compared to Code. However, experience showed that it might not be flexible enough to express more complex parallel algorithms [12]. In these respects, Hence suffers from similar limitations (e.g., lack of flexibility and extensibility) as FrameWorks and Enterprise.

2.6.5 Tracs

Tracs [6] was developed at the University of Pisa, and it provides an elegant graphical user interface for developing message-passing parallel programs. It uses sepa-

ration of specifications, similar to the other systems described previously.

Tracs is based on the message passing programming model. In this model, at a given instant, a set of applications may be running, each comprising of one or more tasks. A task can interact with other tasks primarily via a service, a mechanism similar to a remote procedure call. Tasks belonging to the same application can also communicate via a low-level mechanism, either synchronous or asynchronous point-to-point communication via uni-directional channels. Each task is context-insensitive, meaning that its code does not depend on which tasks it interacts with, or which host it is placed on. The languages supported are C, C++ and Fortran.

Application development is composed of two distinct phases: the definition phase and the configuration phase. In the definition phase, the user defines the three basic components of an application: the message model, the task model and the architecture model. A message model defines a template for the structure of a message. An application has a collection of such message models, which have to be identified during this phase. The second component, a task model, is a complete description of a task, starting from the language to be used, ports, services and message models used by the task, etc. The architecture model defines the software architecture of the parallel application in terms of *formal* message and task models. An architecture model defined during this phase can be saved in a user-defined library for latter use.

During the configuration phase, the programmer constructs the complete application from the basic components, either defined during the definition phase or selected from the system libraries or both. The libraries are composed of two parts: the user-defined libraries and the libraries supplied with the environment. If everything is selected from the libraries, the definition phase can be skipped. Separation of the development process into the two distinct phases is exactly the separation of

specifications issue discussed earlier. An elegant graphical user interface aids visual development of an application during both the phases of development.

Patterns in Tracs are actually the architecture models, constructed from formal message and task models, defined during the definition phase of the development. It supports the notion of extensibility of the system, unlike the other systems described previously, by allowing the user to define and save an instantiated pattern for future use. However, Trac's way of realizing extensibility and re-usability has several limitations. First, the complete graphical definition of a pattern from the two basic components, message and task models, has limited scope. For instance, there is no elegant way to define a dynamically created divide-and-conquer tree graphically. Second, the graphical model does not support the creation of some very useful patterns; for instance, a pattern that uses peer-to-peer interaction as in a data-parallel mesh. Thus, an application needing a data-parallel mesh for its solution is generally out of its scope. Third, the patterns that can be defined and saved graphically are non-parameterized, and hence, are not general. For instance, the user can visually instantiate a *5-slave master-slave pattern* and save it for future use, but not a general *master-slave pattern*, which is more elegant and useful as a re-usable component.

2.6.6 DPnDP

DPnDP [65, 66] was developed at the University of Waterloo with an intention to handle two of the major limitations of some of the previous pattern based systems: lack of flexibility and closeness (i.e., non-extensibility). The DPnDP model provides four basic components: nodes, ports, channels and message handlers, from which an application can be developed either textually or graphically. The pro-

gramming model supported is similar to a client-server type model. A node can either be a singleton, containing sequential application code, or a *design pattern*. Design patterns like master-slave, pipeline, and replication were incorporated into the system library and could be used as reusable components. It applies a two step development process, similar to the other systems described previously, and thus separates most of the parallelism related issues from the sequential application code.

Though DPnDP was intended to be a flexible and extensible pattern-based system, where new patterns could be incorporated into the system by a user as per requirements, these intentions were not fulfilled. This failing can be attributed to the following limitations of the model. First, in DPnDP, each pattern is modeled as a server processing requests in first-come-first-served order. This single scheduling technique, based on the client-server paradigm, reduces the generality of the model. Second, although each design pattern is modeled as a server, the approach advocates message passing to access the low-level primitives. The model is quite unclear about the separation between these two types of interactions. Third, and this is a major limitation, although DPnDP provides a methodology for defining a new design pattern for extending the library, the definition only allows one to create the desired structure of the design pattern. The provision of interaction primitives among the various computing modules, which brings in the behavior, is not included in the methodology. Fourth, the DPnDP model does not specify the constituents of a pattern and its interfaces with other patterns. With this lack of information, the designer is uncertain regarding how to design and add a new patterns to the system without affecting the rest, which substantially hampers the extensibility of the approach.

Although DPnDP did not meet its goals, it did provide a good learning experi-

ence and also set up the stage for further research into the area in order to overcome some of its major limitations.

2.6.7 Archetypes

Concurrent program archetypes was a project at the California Institute of Technology [16, 17]. As mentioned in [17] "*A concurrent program archetype aids in the development of reliable, efficient, parallel applications with common computation/communication structure by providing development methods and code-libraries specific to that structure*". An archetype is a collection of three components: a method of problem solving for a restricted class of problems, a program design strategy associated with this method, and a collection of tutorial example application programs in different languages and run-time systems, and on different target architectures for each application.

The work on Archetypes so far has not proposed any software tool or model for parallel application development. Rather, the emphasis is on understanding the commonly used parallel structures via example implementations and documentation, so that the knowledge can be conveyed in a systematic manner.

2.6.8 Model Programming

The work by Per Brinch Hansen [9, 10] focuses on similar ideas in the domain of scientific computing. In the author's words, a programming paradigm is a class of algorithms that solve different problems but have the same control structure. The work studies a number of such paradigms, for instance: all-pairs pipeline, the multiplication pipeline, the divide and conquer tree, the divide and conquer cube, parallel Monte Carlo trials and the cellular automata.

For each paradigm, a general program is written that defines the common control structure. Such a program is called an *algorithmic skeleton*, *a generic program* or a *program template*. Such a program contains a few unspecified data-types and procedures that vary from one application to another. A *model program* is obtained by replacing these data-types and procedures from a sequential program that solves a specific problem. Thus, a *model program* has a parallel component that implements a paradigm and a sequential component for a specific application. This approach is exactly the issue of *separation of specifications* discussed before. Description of each paradigm includes example applications.

Similar work on algorithmic paradigms and skeletons, more from the functional programming perspective, is a major focus point of the so called *algorithmic skeletons* research group, which is briefly discussed next.

2.6.9 Algorithmic Skeletons

One of the pioneering works in the area of parallel algorithmic patterns is the Ph.D. research by Murray Cole [18] at the University of Edinburgh. One of these patterns is already illustrated in the beginning of this chapter under section 2.3. Parallel algorithmic patterns in Cole's work, and other similar works, are often represented as high-order polymorphic functions, which are best represented using the various functional programming languages [29]. However, as Cole's work suggests, conventional imperative languages (specially those that support functions as parameters) can also be used for realizing higher-order polymorphic functions in the form of *program-* or *procedural-templates*.

Logic programming languages, in the form of predicates and associated clauses, can also be used to mimic the functionalities of high-order functions. In this case,

program execution consists of deciding whether the outermost predicate is true, given its arguments as clauses and their definitions. With this approach, it is even possible to specify the outermost predicate with unbound arguments. The purpose of program execution in this case is to find bindings to the unbound arguments allowing the predicate to be satisfied, or to determine if no such bindings exist.

The other pioneering work by Ian Foster et al [22] takes a similar approach from the logic programming viewpoint, in their high-level representation of algorithmic skeletons. This work uses the concurrent logic programming language Strand [23] in its representation of algorithmic skeletons. A Strand program consists of a collection of guarded rules, where each rule resembles a predicate and a set of associated clauses, plus some added features. Program development in Strand facilitates source-to-source-transformation, which is one of the features of this approach. This facility allows a programmer to develop an application in a form that is convenient to him, and is automatically transformed to a form convenient to the system so that the resultant application can be linked with the system library. Another feature is the possibility for creating new skeletons, from partly existing ones and partly new skeleton-code, using composition rules without rewriting all from scratch.

In the work by Darlington et al [21], the authors follow a path similar to Cole's work for imperative languages. As a distinction, algorithmic skeletons in this work are formulated in a non-strict functional programming language called Haskell [39]. Unlike Cole's skeletons, where the programmer is completely unaware of the underlying architecture, skeletons in this work are augmented with documents regarding resource allocation issues for each skeleton/machine pair. The resource allocation issues are addressed explicitly during implementation in an interactive manner. Accordingly, the programming model here is semi-explicit as compared to the Cole's implicit approach.

Of late, there are several variations of research works in this direction. A comprehensive survey and comparison of the various algorithmic skeletons formulated by some of the well known researchers in this area can be found in [14].

2.7 Limitations of the Existing Pattern-Based Approaches

Although the idea of design- and implementation-level patterns holds significant promise, in practice however, most of the pattern based systems mentioned before face some or all of the following severe limitations:

1. *Limited flexibility*: In most systems, the user is often restricted by a limited set of pre-defined patterns and hard-set restrictive rules. Often, if some desired pattern is not supported by the system, the user has no alternative but to quit the idea of using the particular approach altogether.

2. *No extensibility*: Most systems are hard-coded with a limited and fixed set of patterns, and often there is no method for adding new patterns to the system whenever need arises. This type of closeness hampers the usability of the approach.

3. *Ad hoc patterns*: Most systems support an ad hoc set of parallel patterns, without providing any canonical definition of a pattern. This omission has serious adverse implications, restricting the user's ability to develop applications using pattern-composition, and also in designing new patterns which need to work in conjunction with the existing ones.

4. *Language*: Many of the pattern-based systems are based on new languages or language extensions. The work on algorithmic skeletons is based on abstract mathematical concepts and is best suited for implementation using the various functional and logic programming languages. However, adoption of such approaches by the main-stream parallel programming community becomes an issue, where the conventional languages (like Fortran, C, C++ and Java) and programming models remain the preferred choices for the majority of developers. Moreover, adoption of new languages and/or programming paradigms also directly affects other important issues like software reusability and maintainability.

All these factors severely restrict the usability of a particular approach. A detailed discussion on these shortcomings can be found in [63]. More comparisons with some of the related works appear towards the end of the thesis.

2.8 A Generic Model for Pattern-Based Parallel Computing

The idea of design- and implementation-level patterns in parallel computing holds significant promise and is an active area of research at this moment. However, most of the current pattern-based approaches suffer from severe limitations, some of which include: lack of flexibility, zero extensibility, ad hoc patterns hindering pattern-composition, and language related limitations.

In contrast to the previous approaches, this research proposes a generic pattern-based model for the design and development of parallel applications. The model is generic because it can be described in a manner independent of patterns and

applications. The model is based on the message-passing paradigm, which makes it particularly suited for a network of workstations and PCs. It combines the flexibility of a low-level, MPI-like message-passing environment with the benefits of high-level parallel patterns, which provides the necessary flexibility in application development. The generic model, as opposed to being ad hoc, enhances usability. As is discussed later, the generic model also contributes towards extensibility. As it turns out, the model can be ideally implemented using object-oriented techniques. An object-oriented and library-based implementation of the model in C++, using MPI as the layer underneath, has been completed without necessitating any language extension. The object-oriented and library-based approach, in conjunction with the generic definition of a pattern, facilitates extensibility.

The next two chapters discuss the model and its present object-oriented implementation. The subsequent chapters further elaborate on issues like the applicability of the model in the realization of the various parallel patterns mentioned earlier, issues ranging from flexibility and extensibility to the various software engineering aspects of the model, and finally performance related issues of the present implementation.

Chapter 3

Parallel Architectural Skeletons

This chapter introduces the Parallel Architectural Skeleton Model (abbreviated PASM). Latter in the chapter, the model is further elaborated with the help of a few examples. Implementation issues of the same examples, from the perspective of the PASM system, are discussed in the next chapter. The model is discussed next.

3.1 The Model

A *parallel architectural skeleton* [31–33] is a set of attributes that encapsulate the structure and the behavior of a parallel pattern in an application independent manner. These attributes are generic for all patterns. As is described later in this chapter, many of these attributes are parameterized where the value of a parameter depends on the needs of an application. Some of these parameters are statically configurable (i.e., at compile time) while the others are dynamic (i.e., run-time configurable). User extends a skeleton by specifying the application-dependent

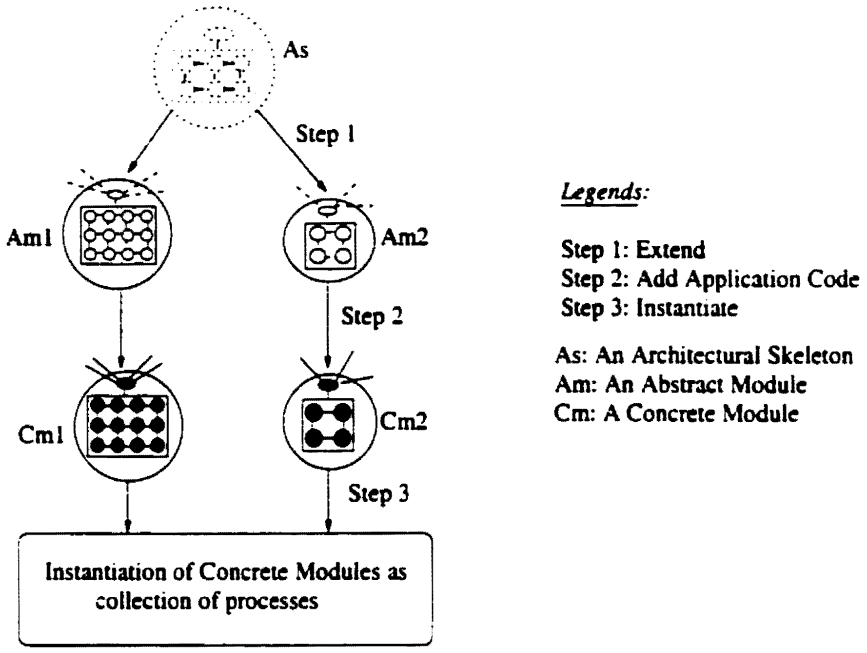


Figure 3.1: Application development using architectural skeletons

static parameters, as needed by the application at hand.

Figure 3.1 approximately illustrates the various phases of application development using parallel architectural skeletons. As shown in the figure, different extensions of the same skeleton can result in somewhat different *abstract parallel computing modules* (abbreviated as an *abstract module*). An abstract module is yet to be filled in with application code. Once an abstract module is supplied with application code, it results in a *concrete parallel computing module* (abbreviated as a *concrete module* or simply a *module*). A parallel application is a systematic collection of mutually interacting, instantiated modules.

An abstract module inherits all the properties associated with a skeleton. Besides, it has additional components that depend on the needs of a given application. In object-oriented terminology, an architectural skeleton can be described as the

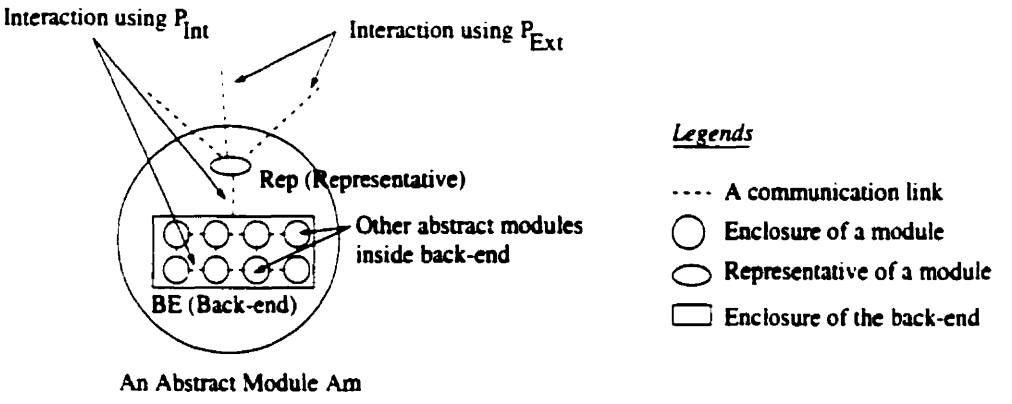


Figure 3.2: Structure of an abstract module

generalization of the structural and behavioral properties associated with a particular parallel pattern. An abstract module is an application-specific *specialization* of a skeleton.

Figure 3.2 diagrammatically illustrates the anatomy of an abstract module (in this case, the module extends the data-parallel architectural skeleton designed for 2-D mesh topology). The various attributes associated with a skeleton (and subsequently inherited by an abstract module and a module) are explained next.

3.1.1 A formal description of the model

Definition 1.1: An *architectural skeleton*, As, is an application-independent abstraction comprising of the following set of generic attributes, $\{Rep, BE, Topology, P_{Int}, P_{Ext}\}$. An *abstract module* is an application-specific extension of a skeleton. Let Am be such an abstract module that extends the skeleton, As. The various attributes inherited by Am (from As) are described in the following:

- *Rep* is the representative of Am. When filled in with application code, *Rep* represents the module in its action and interaction with other modules.

- BE is the back-end of Am. Formally, $BE = \{Am_1, Am_2, \ldots, Am_n\}$, where each Am_i is itself an abstract module. The notion of modules inside another module results in a tree-structured hierarchy. Am, at the root of this tree, is the *parent* and each Am_i is its *child*. Modules Am_i and Am_j belonging to the same back-end are *peers* of one another.

- *Topology* is the interconnection-topology specification of the modules inside the back-end (BE), and their connectivity specification with *Rep*.

- P_{Int} is the internal communication-synchronization protocol of Am and its associated skeleton, As. The internal protocol is an inherent property of the skeleton, and it captures both the parallel computing model of the corresponding pattern and the topology. Formally, P_{Int} is a set of communication-synchronization primitives. Using the primitives inside P_{Int}, the representative of Am can interact with the modules in its back-end, and a module in the back-end can interact with its peers.

- P_{Ext} is the external communication-synchronization protocol of Am. Formally, it is defined as a set of primitive commands. Using the primitives inside P_{Ext}, Am can interact with its parent and the peers. Unlike P_{Int}, which is an inherent property of the skeleton, P_{Ext} is adaptable. That is, Am adapts to the context of its parent by using the internal protocol of its parent as its external protocol. Formally, $(P_{Ext})_{Am} = (P_{Int})_{ParentAm}$.

Though an abstract module is an application specific specialization of an architectural skeleton, it is still devoid of any application code. A user writes application code for an abstract module using its communication-synchronization protocols, P_{Int} and P_{Ext}. A code-complete abstract module is called a *concrete parallel com-*

puting module (abbreviated as a *concrete module* or a *module*). A concrete module can be formally defined as follows:

Definition 1.2: (1) An abstract module with no children (i.e., an empty *BE*) becomes concrete as soon as its representative, Rep, is filled in with application code. (2) An abstract module with children becomes concrete provided each of its children is a concrete module and its own representative is filled in with application code. A parallel application is a hierarchical combination of mutually interacting concrete modules.

As is mentioned before, the notion of parent-child relationships among modules results in a tree-structured hierarchy. A parallel application can be viewed as a hierarchical collection of modules, consisting of a *root* module and its children forming the sub-trees. This tree is called the *HTree* associated with the application. The hierarchy can be formally defined as follows:

Definition 2: (a) Let us consider a module M, either abstract or concrete. Let *Rep* be its representative and $BE = \{M_1, M_2, \ldots, M_n\}$ be its back-end. The hierarchy associated with M is denoted as HTree[M] and is recursively defined as the set, HTree[M] = {*Rep*, HTree[M_1], HTree[M_2].....HTree[M_n]}. In other words, *Rep* of M is at the root of the tree, and the modules in the back-end form the sub-trees. (b) Let the module M form the root of an application's hierarchy. In that case, the *HTree associated with the application* is the same as HTree[M]. The application becomes *complete* as soon as M becomes a concrete module (also refer to Definition 1.2).

Every parallel application is structured as an *HTree*. For instance: (1) in a **Master-Slave** application, which can be implemented using the dynamic replication skeleton, the **Master** module forms the root of the hierarchy and the dynamically

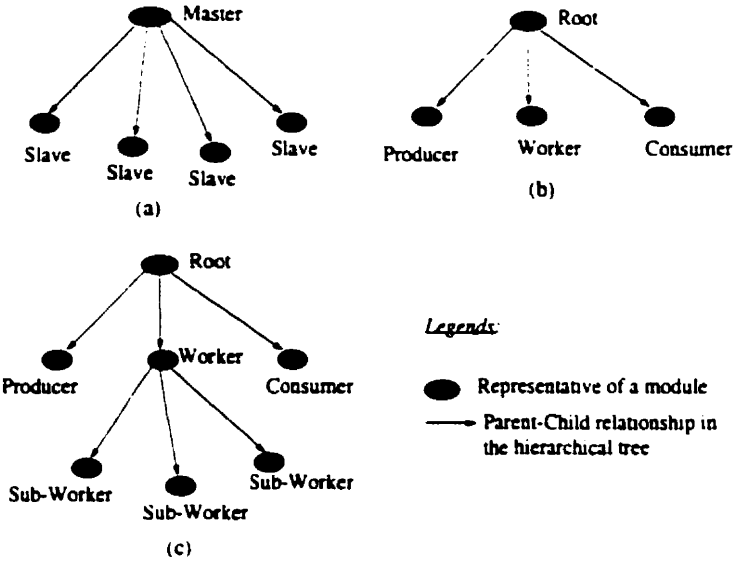


Figure 3.3: Diagrammatic representation of a HTree

replicated children, `Worker` modules, form the sub-trees. (2) In an application consisting of the three modules: `Producer`, `Worker` and `Consumer`, a *compositional module* (i.e., a module which extends the *compositional skeleton*) forms the root of the hierarchy, and its three children (i.e., `Producer`, `Worker` and `Consumer`) form the sub-trees. In either case, a singleton module that has no children is a leaf. HTrees associated with these two applications are illustrated in Figure 3.3(a) and (b) respectively.

Definition 3: As is seen earlier, a parallel application is a hierarchical collection of modules. Each module takes in some inputs from other modules (i.e., its parent and peers), performs some action, and produces some outputs to other modules. It is possible to replace a module M_i with another module M_j, while keeping this replacement transparent to its parent and peers, provided M_j has the same input-output interface and performs the same action as M_i. This type of localized replacement that might aid towards the betterment of a parallel application

is called *refinement*. Refining a parallel application is equivalent to modifying a sub-tree of the associated HTree, without affecting the rest.

For instance, let us consider the **Producer-Worker-Consumer** application, mentioned earlier. Initially, each of the three modules is a singleton module (Figure 3.3(b)). However, **Worker** is found to be very time consuming, and hence, is refined to a dynamic-replication module of identical name. In this case, the **Worker** module dynamically replicates its work-load to **Sub-Workers**, each of which is a singleton module. The corresponding change in the HTree is illustrated in Figure 3.3(c). Note that the modules **Root**, **Producer** and **Consumer** remain untouched by this change.

To summarize, an architectural skeleton is a pure application-independent abstraction. An abstract module contains some application specific components (e.g., the right parameters for topology, the right protocol depending on the current context). A concrete module is an application-specific completion. A hierarchy comprising of only abstract modules represents the overall structure of an application, without application code. From the implementation perspective, such a structure can be compiled and run, however without doing anything useful.

In the rest of the discussion, the parallel architectural skeleton model will be abbreviated as PASM wherever appropriate. The next section illustrates the theory behind the model with various examples.

3.2 Examples

This section exemplifies the theory behind the PASM model in an implementation-independent manner. The next chapter discusses the current object-oriented im-

plementation and re-visits the examples. More examples are presented in chapter 5 which describes the catalog of existing parallel architectural skeletons.

3.2.1 A Graphics Animation Application

Let us consider the graphics animation application [63] already discussed in Chapter 2. As mentioned before, it consists of the three modules: **Generate**, **Geometry** and **Display**. The application takes a sequence of graphics images, called frames, and animates them. **Generate** computes the location and motion of each object for a frame. It then passes the frame to **Geometry**, which performs actions such as viewing transformation, projection and clipping. Finally, the frame is passed to **Display**, which performs hidden-surface removal and anti-aliasing. Then it stores the frame onto the disk. After this, **Generate** continues with the processing of the next frame and the whole process repeats.

Each of **Generate**, **Geometry** and **Display** performs sequential computation (at least, for the time being), and together they form a pipeline. Parallelism is obvious in this case: each pipeline stage can work concurrently with the other two stages and speed-up should be achieved as long as the pipeline remains full.

The *singleton skeleton* is designed for single-process, single- or multi-threaded computation, and hence, it can be extended to create each of the three sequential computing modules: **Generate**, **Geometry** and **Display**. Together they form a pipeline. Either of the *pipeline skeleton* or the *compositional skeleton* can be used to compose the three sequential modules to form the pipeline. In this case, it is decided to use the compositional skeleton due to the fact that some other features of the model (e.g., flexibility) are enhanced by the compositional skeleton and are discussed later in the thesis. The compositional skeleton is used to irregularly

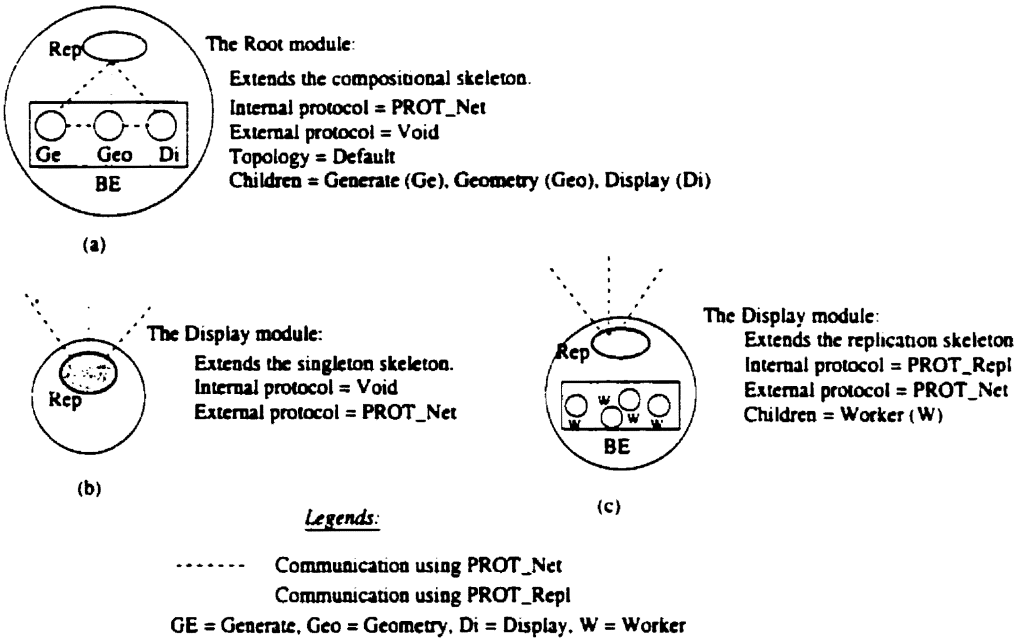


Figure 3.4: Structure of the animation application

compose other modules. By default, the modules composed (i.e., the child modules inside the back-end) are all-to-all interconnected. The internal protocol P_{Int} of the compositional skeleton is PROT_Net = {Send(...), Receive(...) , Broadcast(...), Spawn(...)....}.

After deciding on selecting the appropriate architectural skeletons, the application is structured as follows. A compositional module (initially abstract) named *Root* extends the compositional skeleton and forms the root of the hierarchy (Figure 3.4(a)). The module *Root* extends the compositional skeleton by specifying the following application-specific static parameters associated with the various attributes discussed before: (1) the constituents of its back-end, which in this case are: **`Generate`**, **`Geometry`** and **`Display`**; (2) the topology specification which, in this case, is the default fully-connected topology; and (3) the adaptable external

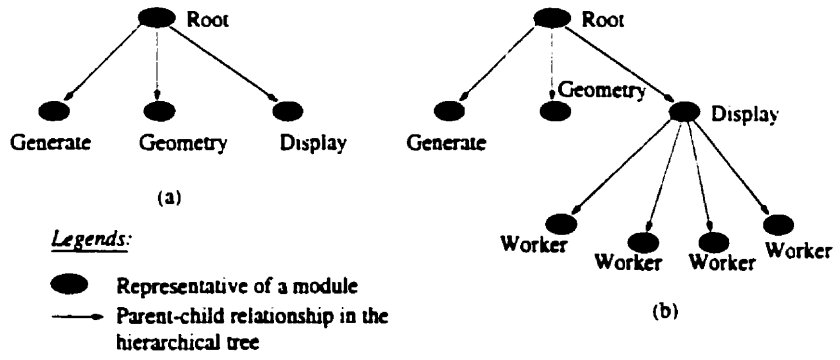


Figure 3.5: HTree representation of the animation application

protocol, P_{Ext}, which is void in this case since *Root* is at the root of the hierarchy and accordingly it has no parent.

Each of the three (abstract) modules **Generate**, **Geometry** and **Display** is formed by extending the singleton skeleton (Figure 3.4(b)). In each case, the following single static parameter needs to be specified while extending the skeleton: the adaptable external protocol, P_{Ext}. In this case, the external protocol becomes PROT_Net, which is the same as the internal protocol of the parent, i.e., the *Root* module. Thus, in one sense, the external protocol is implied by PASM and hence it does not need explicit specification by the user. Note that a singleton module can have no children, and hence, both of its internal protocol, P_{Int}, and the topology attributes are empty.

Figure 3.4(a) illustrates the top-level structure of the application. The corresponding HTree representation is shown in Figure 3.5(a). Figure 3.4(b) illustrates the anatomy of the **Display** module which, in this case, extends the singleton skeleton. A singleton module can have no children and with an empty back-end it forms a leaf in the hierarchy. The empty back-end of **Display** is not shown in the figure.

Until now, nothing has been mentioned about the application code for each

module. As is discussed in the previous section, each abstract module becomes concrete as soon as it is code-complete (refer to Definition 1.2). Discussion about the code-specific parts of the application is postponed to the next chapter, where the implementation issues of PASM are presented.

Refinement

As discussed in the previous chapter, in a typical graphics application, the `Display` module that performs hidden surface removal and anti-aliasing is the most time intensive of the three children modules. This will slow-down the entire pipeline. The best way to resolve this is to distribute the work-load of `Display` to dynamically replicated (i.e., replicated based on load) workers. Consequently, the singleton `Display` module is replaced with another module, of identical name, which extends the *replication skeleton*. In this case, the work-load of the new `Display` module is distributed among dynamically replicated children, i.e., `Worker` modules. Each `Worker` extends the singleton skeleton.

The internal protocol of the replication skeleton is PROT_Repl, which becomes the external protocol of each `Worker`. The external protocol of `Display` remains the same as before, i.e., PROT_Net.

The new `Display` module is illustrated in Figure 3.4(c). The corresponding change in the HTree representation of the application structure is shown in Figure 3.5(b). Note that the rest of the application is unaffected by this change. In fact, that is exactly the definition of refinement (refer to Definition 3).

3.2.2 Jacobi

This example illustrates an application of the PASM model in a parallel implementation of the Jacobi iterative scheme for solving sparse linear systems. Sparse systems are frequently encountered in various scientific applications, for instance: thermodynamics, computational fluid dynamics, electromagnetics, [50, 59]. In this specific application, it is assumed that a square grid with given boundary conditions is used. The algorithm iterates over all grid points, and at each point it calculates certain value (for instance: temperature) associated with the point based on the values of the neighboring grid points. The algorithm repeats until all the values converge. The converged values correspond to the solution of the collection of linear equations, as represented by the sparse (matrix) system. Evidently, nearest-neighbor communication is an essential ingredient of the Jacobi iteration scheme.

The data-parallel mesh is the most appropriate for this application. The given grid is equally partitioned among the mesh-elements. Some suitable mapping algorithm or heuristic method can be applied in this mapping [30]. However, in the case of a square grid, the mapping is quite straightforward. Multiple grid-points are mapped per mesh-element, which determine the granularity of the application (i.e., the ratio of computational time to communication overhead at each mesh-element, between two successive communications). Note that nearest neighbor communication is needed at the inner mesh-boundaries.

Obviously, the architectural skeleton for mesh-structured data-parallel computation is the most appropriate for this application. Accordingly, the root of the hierarchy is formed by the module named `Jacobi`, which extends the data-parallel skeleton (refer to Figure 3.6(a)). As shown in the figure, its topology parameter is

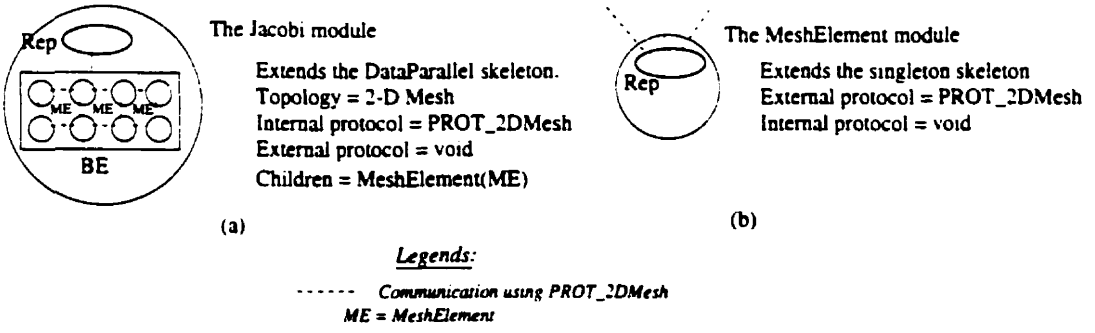


Figure 3.6: Structure of the Jacobi application

selected as 2-D mesh and the corresponding internal protocol is PROT_2DMesh. Its child module is named `MeshElement`. Identical copies of `MeshElement` constitute the back-end of the `Jacobi` module (this is one of the properties of the data-parallel architectural skeleton, associated with its programming model) and together they form a mesh-structured topology.

As is shown in Figure 3.6(b), each `MeshElement` module extends the singleton skeleton. However, it can be refined to any other module based on an application's needs. The adaptable external protocol of the `Jacobi` module is void, since it is at the root of the hierarchy. The adaptable external protocol of each `MeshElement` is PROT_2DMesh, which is the same as the internal protocol of its parent.

After structuring the application this way, each (initially abstract) module is filled in with application code. Code-segments of the Jacobi implementation are illustrated in the next chapter.

3.2.3 Divide and Conquer

The divide and conquer pattern is discussed in the previous chapter. Of all the patterns in parallel computing mentioned earlier, divide and conquer is an interesting

one because realization of this pattern inside PASM is recursive.

As a property of the divide and conquer skeleton, a module extending the skeleton can have multiple copies of itself as its own children. Accordingly for a divide and conquer module, the static parameter that specifies its children is implicit. Each module corresponds to a node of the divide-conquer tree, as discussed in the previous chapter. The internal protocol of each module is PROT_DivideConquer. Though the parent and child modules are identical statically (i.e, at compile time), some of their dynamic characteristics differ. For instance, as is evident from the previous chapter, the root module, a leaf module and an inner module have different functionalities. How does a module dynamically identify itself? This role is played by the primitives inside PROT_DivideConquer (refer to the next chapter).

The divide and conquer tree supported in this model is dynamic in nature, which provides the maximum flexibility to the user. The width of the tree could be either static or dynamic (i.e., run-time configurable), and it is up to the user to make the appropriate selection. A static width corresponds to the fixed number of children each module can have (other than the leaf modules). The height of the divide and conquer tree is dynamic, which implies that whether a module can have further children or not is determined at run-time (based on whether the base condition is satisfied or not, as is discussed in the previous chapter).

In this example, a divide and conquer application of static width three is discussed. The width corresponds to the fixed number of children that each module can have. As is stated before, though the parent and child modules are statically identical, the root module corresponding to the root of the divide and conquer tree has slightly different characteristics than a non-root module. Some of these differences are illustrated in Figure 3.7. Figure 3.7(a) corresponds to the root module (root of the divide and conquer tree, as well as the root of the hierarchy in this

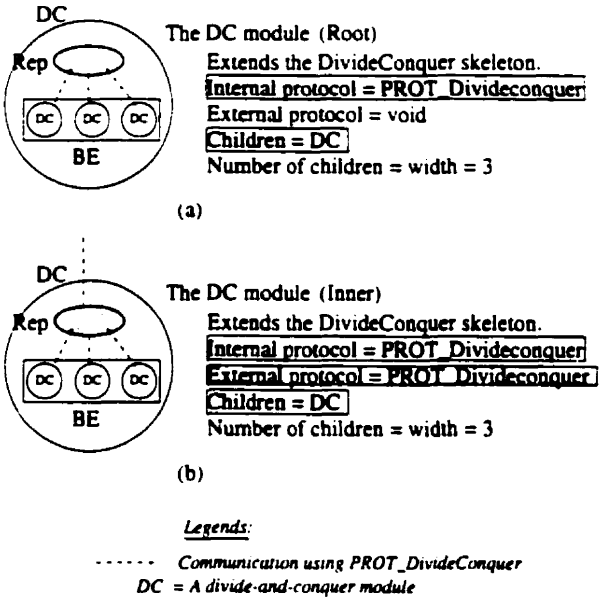


Figure 3.7: Structure of a stand-alone divide and conquer application

application), while Figure 3.7(b) corresponds to an inner module. All the implicit static parameters are highlighted as shaded areas. Note that the external protocol of the root module is void, whereas that of a non-root module is the same as its internal protocol, PROT_DivideConquer. A leaf module has no children and this information is known only at run-time. A leaf module is not illustrated separately. Code segments of a divide and conquer application are illustrated in the next chapter.

3.3 Summary

A pattern-based model for parallel application design and development has been presented. The Parallel Architectural Skeleton Model (abbreviated PASM) is generic because it can be described independent of patterns and applications. The model

is based on the message-passing paradigm which makes it particularly suited for a network of workstations and PCs. A majority of the frequently used patterns in parallel computing are realizable within the frameworks of the model. Some of these patterns and relevant applications are discussed to illustrate the idea behind the model. More examples on patterns and related applications follow in subsequent chapters. Implementation issues of the model are discussed in the next chapter. Other important issues associated with the model and its implementation, e.g., flexibility and extensibility, are discussed in a later part of the thesis.

Chapter 4

An Object-Oriented Implementation

This chapter discusses an implementation of PASM. When the model was originally designed, there was no specific implementation strategy in mind. However, later on, it was realized that PASM is an ideal candidate for object-oriented style design and implementation. Recently, an object-oriented and library-based implementation of PASM has been completed in C++, without necessitating any language extension. Together with the performance measures discussed in chapter 6, the implementation demonstrates the practical feasibility of the model. The key implementation-features are discussed next.

4.1 Basic Implementation Features

The current implementation of the PASM system uses C++ (SUNCC Compiler V4.1). The system is built on top of MPI [35]. There are several vendors who

are working towards the implementation of the MPI standard (presently 2.0), as proposed by the MPI forum [2]. The current implementation of the PASM system uses LAM 6.1 [1], initially developed at the Ohio Supercomputing Center and now maintained and extended at the University of Notre Dame, USA. LAM (Local Area Multicomputer) is an MPI programming environment and development/debugging system for heterogeneous computers on a network. It implements the complete MPI-1 standard and many of the MPI-2 features.

A textual user interface helps the user in various stages of application development. Application code written using the textual interface is parsed by a Perl-script [73] to expand to C++ code, which is subsequently compiled and linked with the skeleton-library to produce the executable. As is illustrated later, the use of the textual interface is not a language-extension, but merely an optional feature that helps the user to skip certain laborious and often monotonous steps in the development process. If desired, the user can bypass this phase and directly work in C++.

Other important features of the current implementation include: (1) use of C++ operator-overloading to implement certain primitive operations inside protocol classes, e.g., Send(...), Receive(...) operations, inside PROT_Net. (2) Implementation of automatic data-marshaling and un-marshaling mechanisms whereby the data attributes of an object, user- or system-defined, can be marshaled, shipped over a communication link and then un-marshaled without the usual hassles of data packing and un-packing as in MPI.

The discussion begins with various examples, including the ones discussed in the previous chapter, illustrating the use of the textual interface, its parser, and the few other steps involved in application development. Subsequently, more subtle issues related to some of the implementation details are covered.

4.2 The Textual User Interface: Examples

This section exemplifies a user's perspective of applying the system in implementing various applications. It is assumed that the user is thoroughly familiar with the PASM model discussed in the previous chapter. However there is no requirement for the user to have any knowledge about the underlying implementation. Some useful implementation related issues meant for an advanced user, i.e., who might want to extend the system, are discussed in the subsequent sections.

A simple sequential application is illustrated next.

4.2.1 Hello world

This first example does nothing more than printing the string "Hello World". However, it demonstrates some important features of the PASM model, its implementation and the current textual user interface. As discussed in the previous chapter, the singleton skeleton is designed for single-process and single or multi-threaded computation, and is the most appropriate for this example.

Figure 4.1(a) illustrates a user's implementation of the application using the current textual interface. The (initially abstract) module, **MyModule**, extends the singleton skeleton. *Rep* is the representative of **MyModule**. The representative *Rep* is initially empty, which corresponds to an abstract module. Filling in of *Rep* with application code results in the concrete module, **MyModule**, as shown in the figure (also refer to Definition 1.2 in the previous chapter). As a property of the singleton skeleton, the back-end of **MyModule** is empty, and hence, the internal protocol, P_{Int}, is void.

The application code written using the textual interface is parsed and expanded

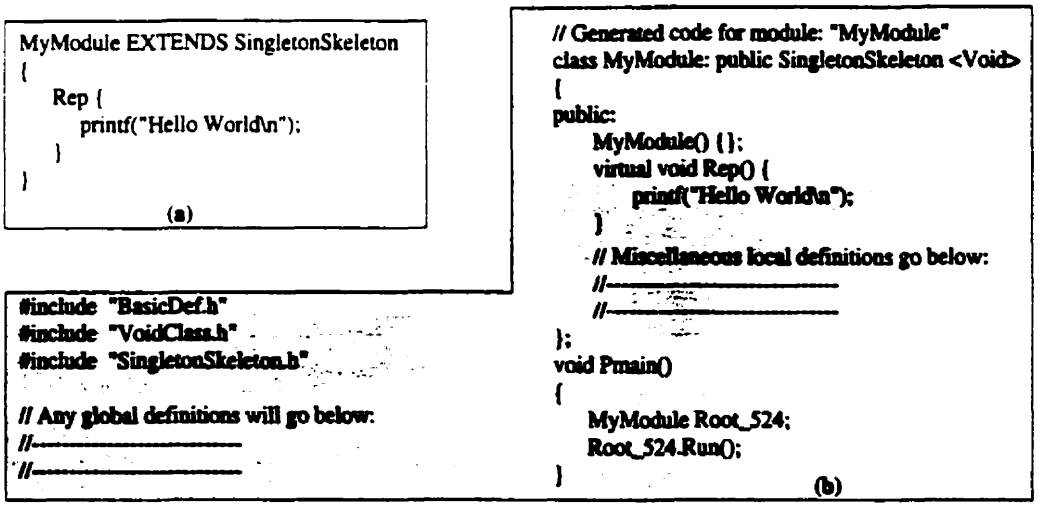


Figure 4.1: Hello World

by a Perl-script to generate the C++ file: Pmain.cc. Figure 4.1(b) illustrates the automatically generated file. Pmain.cc. which is subsequently compiled and linked with the skeleton library to generate the executable. As is evident here. the user can directly develop the application code in C++. The textual interface and its parser merely reduce some of the extra work. which are evidently more pronounced in the examples that follow. Being a stand-alone module. the external protocol. P_{Ext}. of `MyModule` is also void. which is specified as the template parameter *Void* in the generated code.

4.2.2 The graphics animation application

Let us consider the graphics animation application discussed in the previous chapter. The application consists of the three modules: `Generate`. `Geometry` and `Display`. It generates a sequence of graphics images. called frames. and animates them. `Generate` computes the location and motion of each object for a frame. It

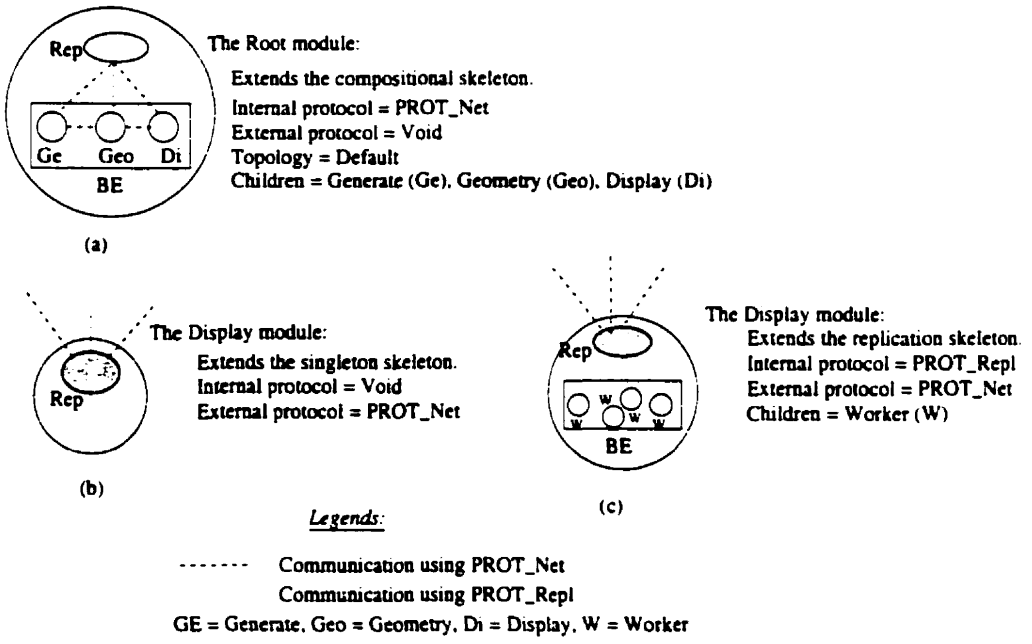


Figure 4.2: Structure of the animation application

then passes the frame to **Geometry**, which performs actions such as viewing transformation, projection and clipping. Finally, the frame is passed to **Display**, which performs hidden-surface removal and anti-aliasing. Then it stores the frame onto the disk. After this, **Generate** continues with the processing of the next frame and the whole process repeats.

One way of structuring the application is illustrated in the previous chapter. For convenience to the reader, Figure 4.2 is repeated here from the previous chapter. The application uses the compositional skeleton and the singleton skeleton. As discussed earlier, the **Root** compositional module (i.e, **Root** extends the compositional skeleton) forms the root of the hierarchy. The three children of **Root** are **Generate**, **Geometry** and **Display**, and they form the subtrees. Each of the three children is initially a singleton module, and hence, is a leaf of the hierarchy.

The following code-segments illustrate one way of implementing the application using the textual interface. The internal protocol (i.e., P_{Int}) of the compositional skeleton is PROT_Net = {Send(...), Receive(...) . Broadcast(...), Spawn(...)....}. Accordingly, PROT_Net becomes the external protocol (i.e., P_{Ext}) for each of the three children.

`GenerateGeometry` and `GeometryDisplay` are user defined classes whose data attributes can be marshaled, shipped over a communication link and then un-marshaled, without the usual hassles of data packing and unpacking. Their constituent data members are either system defined wrappers of standard data-types or other user defined marshal-able types. The example also illustrates the use of C++ operator overloading as an alternative way of defining certain primitive operations inside PROT_Net (e.g., Send(...), Receive(...)).

```
//*********************************************************************
GLOBAL {
// Any global definition may go here.

#include "geom.h"
#define MAXIMAGES 120

   // The following defines a marshal-able class.
   class GenerateGeometry : public UType
   {
      Int imageNumber; // "Int" is a System defined marshal-able wrapper
                       // for "int" type.
      ObjTable table;  // "ObjTable" is a marshal-able class defined
                       // in "geom.h"
   public:
      virtual void Marshal() {imageNumber.Marshal(); table.Marshal();};
                             // Marshal "this" object.
      virtual void UnMarshal() {imageNumber.UnMarshal();
                     table.UnMarshal();}; // Un-marshal "this" object
      // Constructor(s) etc. follow...
      ...
   };

   // Another marshal-able class definition.
   class GeometryDisplay : public UType
   {
      Int imageNumber;
      Int nPoly;
      PolyTable table; // "PolyTable" is another marshal-able
                       // class defined in "geom.h"
   public:
```

```
      virtual void Marshal() {imageNumber.Marshal(); nPoly.Marshal();
                              table.Marshal();};
      virtual void UnMarshal() {imageNumber.UnMarshal();
                              nPoly.UnMarshal(); table.UnMarshal();};
      // Constructor(s) etc. follow...
      ...
   }
}

//*************************************************************************
// The "Root" module, which is at the root of  the  hierarchy.
// It has three child modules: Generate, Geometry and Display.
Root EXTENDS CompositionalSkeleton
{
      CHILDREN = Generate, Geometry, Display;
      Rep {
      // The representative code goes here. In this case, the
      // representative of Root has no functionality.
      }
}

//*************************************************************************
// The "Generate" module, which extends the singleton skeleton.
Generate EXTENDS SingletonSkeleton
{
     // A singleton module can have no children.
     Rep {
       // The representative code goes here.
       int image;
       GenerateGeometry Work;
       for (image = 0; image < MAXIMAGES ; image++){
          ComputeObjects (Work);
          Geometry << Work; // A member primitive of the external
          // protocol: PROT_Net. An alternative option is to
          // use: Send(Geometry, Work, context).
       }
     }
     // All local definitions go below:
     LOCAL {
        void ComputeObjects(GenerateGeometry& Work)
        {
        // User code for "ComputeObjects" goes here.
        }
     }
}

//*************************************************************************
// The "Geometry" module.
Geometry EXTENDS SingletonSkeleton
{
     Rep {
        int image = 0;
        GenerateGeometry Work;
        GeometryDisplay Frame;
        for (image = 0; image < MAXIMAGES ; image++){
           Generate >> Work; // A member primitive of the external
               // protocol: PROT_Net. An alternative option is to
               // use: Receive(Generate, Work, context).
           DoConversion(Work, Frame);
```

```
            Display << Frame;
         }
      }
      LOCAL {
      // Local definition of DoConversion(...) goes here.
      }
}

//*********************************************************************
// The "Display" module.
Display EXTENDS SingletonSkeleton
{
      Rep {
         int image;
         GeometryDisplay Frame;
         for (image = 0; image < MAXIMAGES ; image++) {
            Geometry >> Frame;
            DoHidden(Frame);
            WriteImage(Frame);
         }
      }
      LOCAL { // Local definitions of DoHidden(...) and
              // WriteImage(...) go here.
      }
}
//*********************************************************************
```

As discussed in the previous example, the above code is parsed by the Perl-based parser to produce the C++ file, Pmain.cc, which is subsequently compiled and linked with the skeleton library to produce the executable. The following is the skeleton of the automatically generated file, Pmain.cc:

```
include "BasicDef.h"
#include "VoidClass.h"
#include "CompositionalSkeleton.h"
#include "UnaryHandle.h"
#include "PROT_Net.h"
#include "SingletonSkeleton.h"

// The items defined inside GLOBAL are copied in the following without
// any change.
// ***********************************************************
#include "geom.h"
#define MAXIMAGES 120

// The following defines a marshal-able class.
class GenerateGeometry : public UType
{
// Body of the class is copied as it is.
};

// Another marshal-able class definition.
```

```
class GeometryDisplay : public UType
{
// Body of the class is copied as it is.
}
// ************************************************************

// Automatically Generated code for module: "Generate"

class Generate : public SingletonSkeleton <PROT_Net>
{
public:

     class Params
     {
     public:
         HandleBase* h_73;
         HandleBase* h_113;
         Params(HandleBase* _h_73,HandleBase* _h_113) :
                                  h_73(_h_73),h_113(_h_113){};
         Params() :h_73(0),h_113(0){};
     };
     Params p_611;

     Generate(Params _p) : p_611(_p){};

     virtual void Rep() {
        // The representative code goes here.
        int image;
        GenerateGeometry Work;
        for (image = 0; image < MAXIMAGES ; image++){
            ComputeObjects (Work);
            *(p_611.h_73) << Work; // A member primitive of the external
                      // protocol: PROT_Net. An alternative option is to
                      // use: Send(Geometry, Work, context).
        }
     }

     // LOCAL definitions go here:

     void ComputeObjects(GenerateGeometry& Work)
     {
     // User code for "ComputeObjects" goes here.
     }
}

// Automatically Generated code for module: "Geometry"

class Geometry : public SingletonSkeleton <PROT_Net>
{
public:
     class Params
     {
     public:
         HandleBase* h_481;
         HandleBase* h_113;
         Params(HandleBase* _h_481,HandleBase* _h_113) :
                                  h_481(_h_481),h_113(_h_113){};
         Params() :h_481(0),h_113(0){};
     };
```

```
    Params p_910;

    Geometry(Params _p) : p_910(_p){};

    virtual void Rep() {
       int image = 0;
       GenerateGeometry Work;
       GeometryDisplay Frame;
       for (image = 0; image < MAXIMAGES ; image++){
          *(p_910.h_481) >> Work; // A member primitive of the external
                     // protocol: PROT_Net. An alternative option is to
                     // use: Receive(Generate, Work, context).
          DoConversion(Work, Frame);
          *(p_910.h_113) << Frame;
       }
    }

    // LOCAL definitions go here:

    void DoConversion(GenerateGeometry& Work, GeometryDisplay& Frame)
    {
    // User code for "DoConversion" goes here.
    }
}

// Automatically Generated code for module: "Display"

class Display : public SingletonSkeleton <PROT_Net>
{
public:

    class Params
    {
    public:
        HandleBase* h_481;
        HandleBase* h_73;
        Params(HandleBase* _h_481,HandleBase* _h_73) :
                                    h_481(_h_481),h_73(_h_73){};
        Params() :h_481(0),h_73(0){};
    };
    Params p_21;

    Display(Params _p) : p_21(_p){};

    virtual void Rep() {
       int image;
       GeometryDisplay Frame;
       for (image = 0; image < MAXIMAGES ; image++) {
          *(p_21.h_73) >> Frame;
          DoHidden(Frame);
          WriteImage(Frame);
       }
    }

    // LOCAL definitions go here:
    ...
}

// Automatically Generated code for module: "Root"
```

```
class Root : public CompositionalSkeleton <PROT_Net ,Void>
{
     UnaryHandle<Generate, Generate::Params> h_481;
     UnaryHandle<Geometry, Geometry::Params> h_73;
     UnaryHandle<Display, Display::Params> h_113;
public:

     Root() : h_481(Generate::Params(&h_73,&h_113)),
                          h_73(Geometry::Params(&h_481,&h_113)),
                          h_113(Display::Params(&h_481,&h_73)){};

     virtual void Rep() {
        // The representative code goes here
     }

     // LOCAL definitions go here:
};

void Pmain()
{
        Root TopLevel_820;
        TopLevel_820.Run();
}
```

The automatically-generated C++ code-segments shown above suggest that the textual interface and its Perl-based parser significantly reduce the work-load on the part of the user in application development. The textual interface handles many of the implementation- and C++-related, and other laborious and often monotonous details that can easily be automated.

The textual interface could be developed based on the fact that all applications that use the same architectural skeleton(s) follow similar implementation patterns, irrespective of the applications. For instance, applications that use compositional modules as the root of the hierarchy follow identical implementation patterns, as is observed inside the **Root** class (e.g., the **param** class, the unary handles for children, the various C++ templates, and the typical style in using them). Similarly, all singleton modules that are the children of a compositional module have identical constructs (e.g., the **param** class, unary handles for peers, the external protocol PROT_Net as a C++ template parameter). Obviously, for a stand-alone singleton

module, which is at the root of the hierarchy and hence has no peers, the **param** class is missing. A user can familiarize oneself with these implementation patterns via practice and then directly work in C++. Until then, the textual interface takes care of this part.

C++ templates are extensively used for statically specifying the internal and external protocols for modules, as well as for specifying the static parameters for handles. The functionality of a handle is discussed later. The external protocol of the `Root` module is void and is specified as the template parameter, `Void`. More detailed implementation issues and concepts related to the previous code are discussed in the following sections.

Refinement

As is discussed in the previous chapter, the `Display` module that performs hidden surface removal and anti-aliasing is the most time intensive of the three children modules. Consequently, the singleton `Display` module is replaced with another module of identical name that extends the *replication skeleton*. In this case, the work-load of the new `Display` module is distributed among dynamically created replicas, i.e., **Worker** modules (refer to Figures 4.2(c)).

The internal protocol, P_{Int}, for the replication skeleton is PROT_Repl. Consequently PROT_Repl becomes the external protocol for each replicated child **Worker** module. Note that none of the other modules is affected by this change. This type of localized replacement that works towards the betterment of an application is called a *refinement* (refer to Definition 3 in the previous chapter). The change in the user's implementation is illustrated next:

```
// The refined "Display" module.
Display EXTENDS ReplicationSkeleton
{
     //The dynamically replicated children of "Display"
     CHILDREN = Worker;
     Rep {
        int image = 0;
        int success;
        GeometryDisplay Frame;
        for (image = 0; image < MAXIMAGES; image++){
                Geometry >> Frame; // A member primitive
                 // of the external protocol, PROT_Net.
                success = SendWork(Frame); // A member
                 // primitive of the internal protocol PROT_Repl.

                if (!success) {// Do it myself, if not successful in
                               // assigning to a worker.
                    DoHidden(Frame);
                    WriteImage(Frame);
                }
        }
    }
    LOCAL { ... }
}

// Each replicated "Worker" module
Worker EXTENDS SingletonSkeleton
{
     Rep {
        GeometryDisplay Frame;
        ReceiveWork(Frame); // A member primitive of the external
                            // protocol, PROT_Repl.
        DoHidden(Frame);
        WriteImage(Frame);
     }
     LOCAL { ...  }
}
```

The corresponding automatically generated C++ code is illustrated in the following:

```
// Automatically Generated code for module: "Display"
class Display : public ReplicationSkeleton <Worker, PROT_Repl, PROT_Net>
{
public:

     class Params
     {
     public:
         HandleBase* h_355;
         HandleBase* h_79;
         Params(HandleBase* _h_355,HandleBase* _h_79) :
                                        h_355(_h_355),h_79(_h_79){};
         Params() :h_355(0),h_79(0){};
     };
```

```
     Params p_47;

     Display(Params _p) : p_47(_p){};

     virtual void Rep() {
        int image = 0;
        int success;
        GeometryDisplay Frame;
        for (image = 0; image < MAXIMAGES; image++){
                Geometry >> Frame; // A member primitive
                 // of the external protocol, PROT_Net.
                success = SendWork(Frame); // A member
                 // primitive of the internal protocol PROT_Repl.

                if (!success) {// Do it myself, if not successful in
                               // assigning to a worker.
                    DoHidden(Frame);
                    WriteImage(Frame);
                }
        }
     }

     // LOCAL definitions go here:
     ...
}

// Automatically Generated code for module: "Worker"
class Worker : public SingletonSkeleton <PROT_Repl>
{
public:
     virtual void Rep() {
        GeometryDisplay Frame;
        ReceiveWork(Frame); // A member primitive of the external
                            // protocol, PROT_Repl.
        DoHidden(Frame);
        WriteImage(Frame);
     }

     // LOCAL definitions go here:
     ...
}
```

It should be noted that the initial part of the new **Display** class that deals with establishing connection with the peers through the use of handles, through the definition of the **param** class and its subsequent declaration and instantiation, remains identical to the old **Display** class before refinement. This simply reflects the fact that the **Display** module, as seen by its parent and the peers, remains unchanged from before. It is only the internal representation of **Display**, i.e., the sub-tree with **Display** at its root, that has changed.

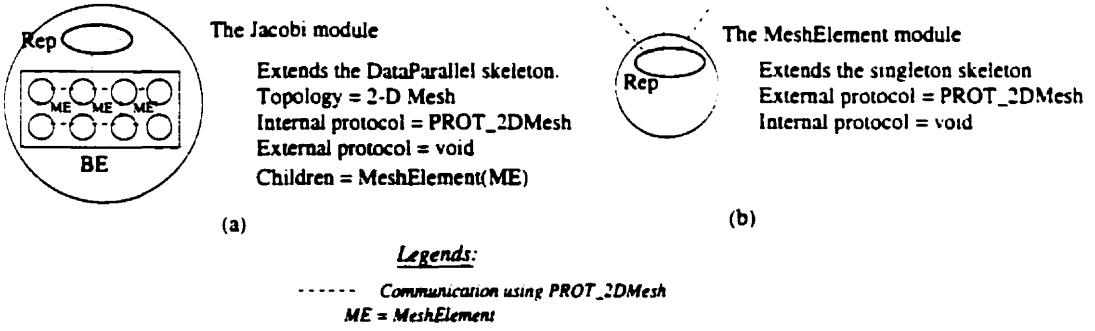


Figure 4.3: Structure of the Jacobi application

The **Worker** class extends the singleton skeleton. The (implicit) external protocol, PROT_Repl, is statically configured as a C++ template parameter. The peers of **Worker** are implicit in this case, which are instantiated copies of itself (as a property of the parent replication module). Accordingly, the "**param** class-definition" part, that is used for establishing connections with the peers, is missing. All these issues are elaborated in the following sections.

4.2.3 Jacobi

The Jacobi iterative scheme and one possible way of structuring the application are illustrated in the previous chapter. It uses the two modules: **Jacobi** and **MeshElement** (refer to figures 4.3(a) and (b), re-produced here from the previous chapter). The **Jacobi** module corresponds to the front-end of a data-parallel mesh. The **MeshElements** constitute the back end.

The following code segments illustrate an implementation of the Jacobi scheme using the current textual interface. For reasons of both simplicity and efficiency, the implementation shown here uses a 1-D mesh. Since Jacobi involves nearest neighbor communications at the mesh boundaries, use of a 1-D mesh reduces the number of

boundary communications to half as compared to a 2-D mesh. The implementation illustrates the use of C++ operator-overloading technique for implementing certain primitive operations inside PROT_1DMesh (for instance: primitives for nearest neighbor communication), and also more on automatic data marshaling and un-marshaling.

```
#include "Grid.h"
...
// ***************************************************
// The "Jacobi" module corresponds to the front-end of an 1-D mesh.
Jacobi EXTENDS DataParallelSkeleton
{
     CHILDREN = MeshElement;
     PROTOCOL = PROT_1DMesh; // In this case, we need to explicitly specify
                             // the internal protocol, since more than one
                             // choice is possible.
     Rep {
          ...
          int N = SetMeshWidth(4) // Set mesh-width to 4. It is a a member of
                                  // PROT_1DMesh. Mesh-width is one parameter
                                  // that can be configured either statically
                                  // or dynamically.
          Grid A(1000,1000); // A 1000 X 1000 marshal-able grid.
          ReadIn (A);
          PartitionGrid (A,N); // Partition the grid row-wise among the
                               // N = 4 children
          CollectResults (A,N); // Collect the results from the children.
          ...
     }
     LOCAL {
     // Definitions of ReadIn (...), PartitionGrid (...), CollectResults (...)
     // and other methods and local variables  may go here (or may be defined
     // globally).
     }
}
// ***************************************************
// Each element of the 1-D mesh.
MeshElement EXTENDS SingletonSkeleton
{
     Rep {
          ...
          int context = ...;
          Grid A;
          ReceiveFromRep(A,context); // It is a member of external protocol,
          // PROT_1DMesh. In this particular case, A is a 252 X 1000 grid.
          // There are two extra rows (i.e., rows 0 and 251) for holding
          // boundary rows from neighboring elements.
          Grid B = A;
          ...
          int lb, ub;
          int nRows = A.Rows();
          int nColumns = A.Columns();
          int myPosition = getMyPosition(); // Get my position in the 1-D mesh.
```

```
                                    // It is a member primitive of PROT_1DMesh.
        int meshWidth = getMeshWidth(); // Get the width of the 1-D mesh. It
                                        // is a member primitive of PROT_1DMesh.
        if (myPosition == 0) lb = 2; else lb = 1;
        if (myPosition == (meshWidth - 1)) ub = nRows - 3; else ub = nRows - 2;

        // MAXITERATIONS is chosen as some reasonable value.

        for (int k = 0; k < MAXITERATIONS; k++){
            if (myPosition > 0) Peer[Left] << A[1]; // Each row of A is a
                                                    // marshal-able object.
            if (myPosition < (meshWidth - 1)) Peer[Right] >> A[nRows - 1];
            if (myPosition < (meshWidth - 1)) Peer[Right] << A[nRows - 2];
            if (myPosition > 0) Peer[Left] >> A[0];

            // The above four statements illustrate communication with peers
            // (in this case, nearest neighbor communication). Different types
            // of communication, including broadcasting to peers, are possible,
            // which are member primitives of PROT_1DMesh. The above statements
            // also illustrate the use of C++ operator-overloading in
            // implementing certain primitive operations. An alternative option
            // is to use functions calls, e.g., SendToLeft(...),
            // ReceiveFromRight(...), SendToOffset(...), etc.

            for (int i = lb; i <= ub; i++){
                for (int j = 1; j <= nColumns - 2; j++)
                  B[i][j] = (A[i][j-1] + A[i-1][j] + A[i+1][j] + A[i][j+1])/4;
            }
            A = B;
        }
        SendToRep(A, context); // A member of external protocol, PROT_1DMesh.
    }
}
// ************************************************
```

The same steps as before are involved in generating the executable, and hence, are not shown here. Performance results for Jacobi are illustrated in a later part of the thesis.

4.2.4 Divide and Conquer

The last example for this chapter illustrates a parallel implementation of the quick sort algorithm [54] using the divide-and-conquer skeleton. The divide and conquer approach is discussed in the previous two chapters. The implicit children of a divide and conquer module are copies of itself. As a result, the module has to dynamically

differentiate the root of the divide-and-conquer tree, an inner node, and a leaf. The primitive command IamTheRoot() inside PROT_DivideConquer (refer below) lets a module dynamically identify itself. Whether a module is an inner node or a leaf is application dependent, and hence, cannot be judged by a primitive (refer to the use of the "Threshold" value in the following application).

The two other primitive operations inside PROT_DivideConquer employed in this application, i.e., PartitionData(...) and CollectResults(...), are used respectively for dividing the data among the children (i.e., copies of itself) and then collecting the results. All primitive methods are commented with a star (*) for ease in identification. The rest of the methods used in the application are application-specific, and are defined either locally or globally.

```
GLOBAL {
#include <fstream.h>
#define Threshold 200

     // The following methods can be defined either locally or globally, it
     // does not matter in this application.
     void InsertionSort(Aint& A)
     {
     // The insertion sort routine is used below.
     }
     void ReadIn(ifstream& infile, Aint& A)
     {
     ...
     }
     void WriteOut(Aint& A, ofstream& outfile)
     {
     ...
     }
}

QSortModule EXTENDS DivideConquerSkeleton
{
     Rep {
         if (IamTheRoot()){ //* It is a member primitive of the internal
                  // protocol, PROT_DivideConquer, which lets a module
                  // dynamically identify whether it is the root of the
                  // divide-conquer tree.*
              ifstream InFile(...);
              ofstream OutFile(...);
              Aint A; // A system defined marshal-able array of integers.
              ReadIn(InFile, A); // Defined globally.
              QuickSort(A); // Defined locally in the following.
              WriteOut(A, OutFile); // Defined globally.
```

```
        return;
    }
    // Else, I am an inner node or a leaf.

    Aint A;
    ReceiveFromParent(A); //* Member primitive of PROT_DivideConquer.*
    QuickSort(A);
    SendToParent(A);      //* Member primitive of PROT_DivideConquer.*
}
LOCAL {
    // The following methods could also be defined globally.
    void QuickSort(Aint& A)
    {
        if (A.getSize() < Threshold)
                InsertionSort(A);
        else {
                int i = 0;
                int end = A.getSize() - 1;
                int j = end - 1;
                int pivot = A[end];

                while (i < j) {
                        while (compare(A[i],pivot) < 0) i += 1;
                        while (compare(A[j],pivot) > 0) j -= 1;
                        if (i < j)
                                Swap(A,i,j);
                }
                Swap(A,i,end);

                // The following information is used to divide A
                // into two partitions. First partition: 0 to i,
                // Second partition: (i+1) to (end-1).
                Aint PartitionInfo(4);
                PartitionInfo[0] = 0;
                PartitionInfo[1] = i;
                PartitionInfo[2] = i+1;
                PartitionInfo[3] = end - i;

                PartitionData(A,PartitionInfo); //* Partition A among
                    // the child modules as based on the information
                    // provided. It is a member primitive of
                    // PROT_DivideConquer.*
                Auser<Aint> datum; //* A 2-D marshal-able array.
                CollectResults(datum); // Collect results from the
                    // child modules. It is a member primitive of
                    // PROT_DivideConquer.*

                // Now merge the results:
                MergeResults (datum, A);
         }
    }
    void Swap (Aint& A, int i, int j)
    {
    ...
    }
    void MergeResults(Auser<Aint>& datum, Aint& A)
    {
    ...
    }
```

```
    }
}
```

The example further illustrates the use of automatic data marshaling and un-marshaling mechanisms. `Aint` is a system defined integer-array type that can be automatically marshaled and un-marshaled. **`Auser<Aint>`** is a marshal-able array of marshal-able integer-arrays (hence, a 2-D marshal-able array).

There is something interesting with the automatically generated code in this case. The following code-skeleton shows relevant pieces of the generated code:

```
...
class QSortModule :public DivideConquerSkeleton <QSortModule,PROT_DivideConquer,
                                                                    Void >
{
public:

        QSortModule() {};
        QSortModule(Void& _v) {};

        virtual void Rep() {
                if (IamTheRoot()){
                ...
                }
                ...
        }
        ...
};
...
```

In the first line of the previous code, the actual value of the first template-parameter to `DivideConquerSkeleton` is the class `QSortModule` itself. The C++ language allows this type of "recursive" parameter passing, which, in turn, facilitates the implementation to strictly conform to the PASM model.

4.3 Implementation Issues

This section discusses the various implementation issues that are the key features of the present object-oriented implementation of the PASM model. These issues are

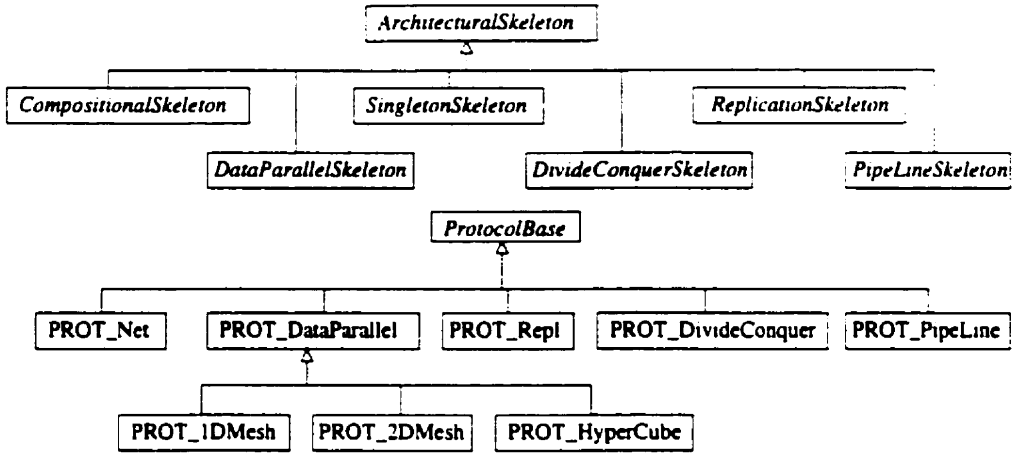


Figure 4.4: High level class diagram behind the design of the skeleton library

also an essential reading for an experienced user who wants to extend the existing system, before start exploring the skeleton-library source code. The implementation of the model is referred to as the PASM system wherever appropriate.

4.3.1 Implementing Architectural Skeletons: Reusability and Extensibility

Figure 4.4 illustrates the high-level class diagram behind the design of the skeleton library. The figure uses the standard UML [8] notation. For simplicity, the figure does not illustrate the relationships between the skeleton- and the protocol-classes. Moreover, the various attributes and the methods associated with each class, and the formal parameters, in the form of templates, associated with each inherited skeleton-class are not shown for a cleaner representation. More detailed UML-representation for an example is illustrated in the next subsection.

From the implementor's or an experienced user's perspective, certain features of the object-oriented design, in conjunction with the generic nature of the PASM

model, favor reuse and extension of the skeleton library. The generic model helps, because it provides a clear picture regarding what the different components of a skeleton are and what their functionalities are going to be (compare it with a totally ad hoc approach). Furthermore, from the model's perspective, each module is an independent entity whose only interface with the outside world is through its representative and the adaptable external protocol. Accordingly, what the outside world sees of the module are only through its actions (i.e., input/output and any observable side effects), without knowing exactly how these actions are carried out internally. In other words, the module acts as a black-box to the outside world. The same arguments apply to the internal structure of the module where its back end might contain copies of itself (for instance: in the divide-and-conquer skeleton) or other modules, with which it can interact only via their representatives. These exclusive features of the model facilitate the design and addition of new skeletons without affecting the existing ones.

Many of the object-oriented features that are supported in C++, for instance: polymorphism (through the use of C++ templates and virtual methods) and inheritance, facilitate the reuse and extension of the existing skeleton library. New classes can be defined by extending existing ones, thus enabling the design and addition of new skeletons and protocols with added functionalities. Completely new skeletons and protocols can be designed by extending the base classes (refer to Figure 4.4). In each case, a collection of pre-existing virtual methods need to be overridden and some new additional methods might need to be defined in order to reflect the characteristics of the newly designed skeleton.

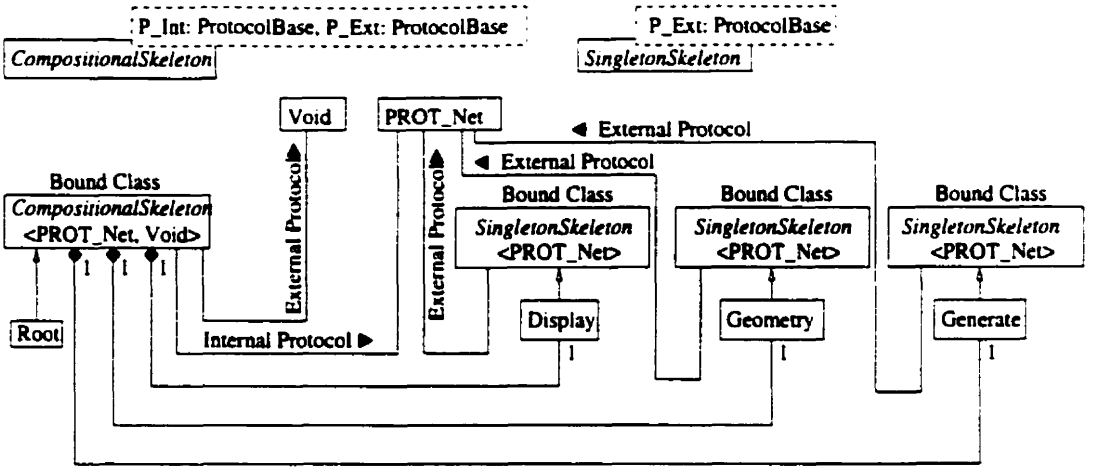


Figure 4.5: High level class diagram for the graphics animation application

4.3.2 The Graphics Animation Application: Revisited

Figure 4.5 illustrates the high-level design in UML notation pertaining to the graphics animation application before refinement, discussed in the previous chapter and detailed in section 4.2.2. As the automatically generated code in section 4.2.2 and figure 4.5 suggest, each architectural skeleton is implemented as a template class, where each template relates to a statically configurable parameter associated with an attribute of the skeleton. For instance, in the case of the compositional skeleton, the two statically configurable parameters are the (implicit) internal protocol, P_Int, and the adaptable external protocol, P_Ext. For design reasons, the other static parameters for the compositional skeleton (for instance: the specification of the children) are not realized as templates. In the case of the singleton skeleton, the only parameter is the adaptable external protocol, P_Ext.

As shown in the figure (also refer to the generated code), each template skeleton class becomes bound as soon as the actual values of the template-parameters are specified. The concrete class `Root` extends the bound (but abstract) compositional

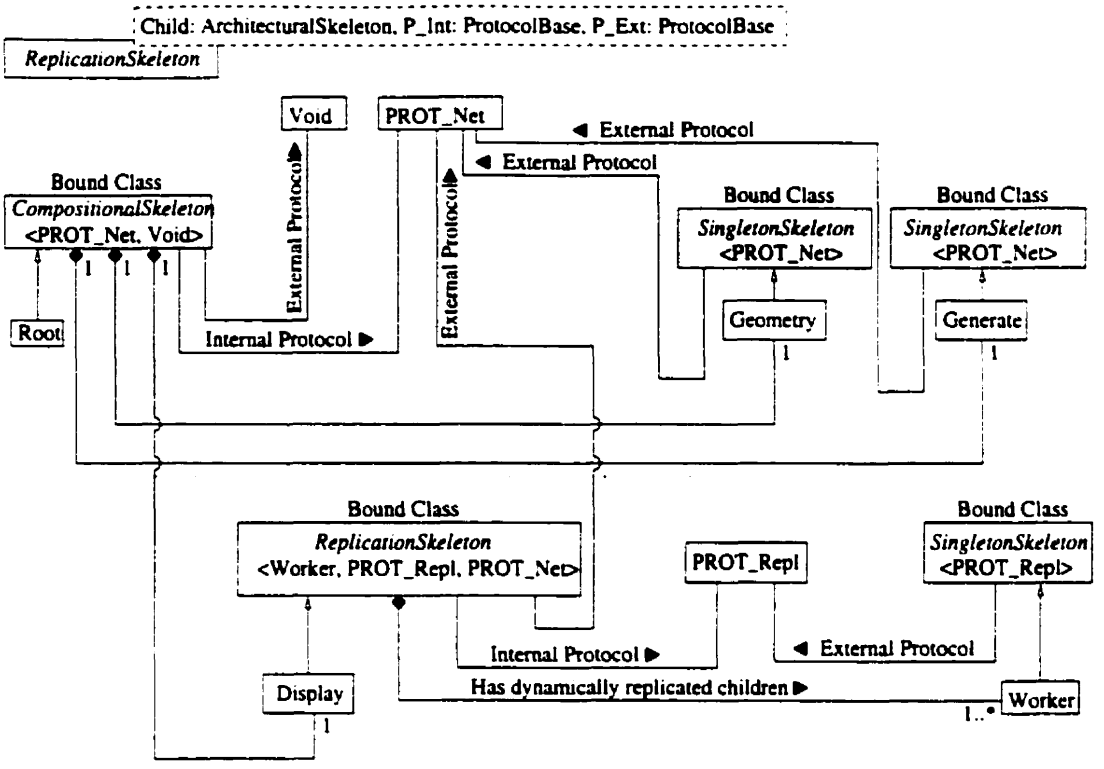


Figure 4.6: High level class diagram after refinement

skeleton class. Similarly, the concrete classes **Generate**, **Geometry** and **Display** extend the bound (but abstract) singleton skeleton classes. Each of **Generate**, **Geometry** and **Display** is contained inside **Root**. The rest of the diagram is self explanatory. The functionality of the handles (i.e., **UnaryHandle**) inside **Root** (refer to the automatically generated code) is discussed towards the end of this section.

Figure 4.6 illustrates the change in the high level design after refinement (refer to the previous chapter and also section 4.2.2). When compared with figure 4.5, it can be seen that the rest the application, other than that involving **Display**, remains intact. The modified part of the design is included inside the dotted rectangle.

As before, the replication skeleton is a template-class. Unlike the compositional

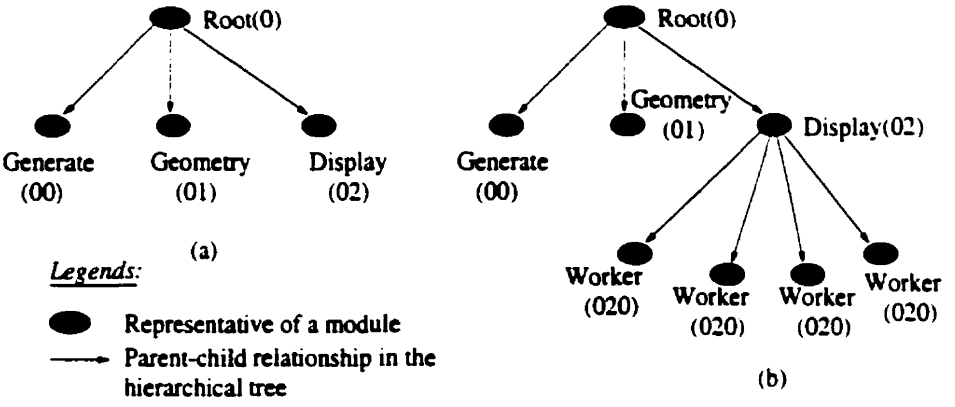


Figure 4.7: HTree and its traversal scheme

skeleton. the replication skeleton has one extra template parameter. that is the child to be replicated (refer to the figure and also to the automatically generated code). Note that the the child is specified here as a template parameter. The child-components of the compositional skeleton are not specified using template parameters because it can have an arbitrary number of different children. The rest of the diagram is self explanatory.

4.3.3 The Dynamic Execution Model

The execution model for the PASM system is SPMD. i.e., each processor in the processor-cluster loads and executes the same file. which results in major savings in terms of management of source. object and executable files. Consequently. each process falls through the same *HTree* associated with the application. starting at the root of the tree. Figures 4.7(a).(b) illustrate the *HTree* associated with the graphics-animation application before and after refinement respectively.

A node of a *HTree* corresponds to the representative of a module (refer to Definition 2 in the previous chapter). Each process is responsible for executing

exactly one node of the tree, that is: there is a one-to-one correspondence between a process and a representative node. A process starts at the root of the hierarchy and then traverses down the tree to its designated node.

How does a process determine the path to traverse? This is achieved as follows: each process is dynamically assigned an identification string (by its parent node based on its own identification string), by following a unique labeling scheme. As a process traverses down the tree, it dynamically calculates its path, by following the same scheme. The process traverses down a specific path of the tree, if and only if the already calculated path is a substring of its assigned identification string. When the calculated path matches the identification string, the process is at its designated node and thus it can execute the code pertaining to that node.

In figure 4.7, the string in parentheses beside each node is the identification string that determines which process executes the node. The dynamically replicated `Workers` after refinement are all identical. Therefore they execute the same code and have the same identification string. Whenever processes with the same identification string have to identify their relative positions with respect to their peers (for instance: inside a 2-D mesh, or for the same level nodes inside a divide-and-conquer tree), they use MPI's internal "rank" mechanism for finding their positions inside a communicator group [1].

All of the previous issues are completely hidden from the user. In fact, for instance, a user follows the general structure as illustrated by the examples in this chapter, and writes an application with the perspective that one is dealing with individual modules, rather than with individual processes. Without any further aid from the user, the dynamic execution model makes it possible for a process to execute the code segment pertaining to a given module.

4.3.4 Mechanisms for Constructing the HTree

The previous discussion outlined the mechanism that allows a process to traverse down the HTree to its designated node. This subsection discusses some of the key implementation aspects of the traversal scheme. The discussion starts with the basic implementation-mechanism for constructing the hierarchy. For convenience, the discussion is presented from the perspective of the graphics animation application discussed before. Relevant pieces of the generated code (refer to section 4.2.2) are re-inserted in the following:

```
...
...
class Root : public CompositionalSkeleton <PROT_Net ,Void>
{
     UnaryHandle<Generate, Generate::Params> h_481;
     UnaryHandle<Geometry, Geometry::Params> h_73;
     UnaryHandle<Display, Display::Params> h_113;
public:

     Root() : h_481(Generate::Params(&h_73,&h_113)),
                          h_73(Geometry::Params(&h_481,&h_113)),
                          h_113(Display::Params(&h_481,&h_73)){};

     virtual void Rep() {
        // The representative code goes here
     }

     // LOCAL definitions go here:
};

void Pmain()
{
        Root TopLevel_820;
        TopLevel_820.Run();
}
```

The mechanism for constructing the hierarchy is through the conditional construction of objects of type "architectural skeleton" inside another object of the same type. For instance, the objects of type **Generate**, **Geometry** and **Display** are (conditionally) constructed inside an object of type **Root**. Each of **Generate**, **Geometry** and **Display** may, in turn, contain other objects inside it (in fact, **Display** contains objects of type **Worker** after refinement). Also refer to figures 4.5, 4.6 and 4.7.

The decision on the part of a process regarding whether to construct an object inside the context of another object is analogous to deciding whether or not to traverse a specific path down the hierarchy (refer to the previous subsection). This decision is taken inside the handles, as shown by the objects of type `UnaryHandle` in the previous code. In other words, a handle serves as a locking-unlocking mechanism that allows only a selected number of processes to traverse down the hierarchy, based on the previous string-based traversal scheme, while blocking the rest. An object of type `UnaryHandle` contains, besides other information, the reference to the created "architectural skeleton" object. Similarly, there are handles of type `GroupHandle` (for instance: used inside the data-parallel skeleton), where each handle is associated with a group of identical "architectural skeleton" objects.

Inside the procedure `Pmain()` in the previous code, a call to the hidden method `Run()` associated with `Root` is made. This call causes each process to start traversing down the hierarchy, starting at the root. There is also a `Run()` method associated with each handle type. The `Run()` of `Root` conditionally calls the `Run()`s of the three handle-objects. The `Run()`s of the handles, in turn, conditionally call the `Run()`s of their associated "architectural skeleton" objects, and thus this sequence repeats.

As an example, let us consider the case of the process assigned with identification string "02", that is meant for executing the representative of `Display` (refer to the previous subsection and also Figure 4.7). As it traverses down from `Root`, it temporarily calculates its path. Inside `Root`, it calculates its temporary path as "0". Since it is a substring of its assigned string "02", it is supposed to traverse down the hierarchy further. So, the process calls the `Run()` method associated with the handle for `Generate` (that is: the handle `h_481` in the previous code), where it calculates the temporary path as "00". Since it is not a substring of "02", the

process is not supposed to traverse down further this path, and hence, it does not create the associated object, **Generate**. The situation is the same for the handle associated with **Geometry**, where the path is calculated as "01". Finally, inside the handle for **Display**, the temporary path is calculated as "02", which matches with the assigned string for the process. Hence the object, **Display**, is created inside the handle and the process starts executing its representative.

After refinement, **Display** has to further spawn identical **Worker** modules. The corresponding processes are dynamically assigned their identification strings based on its current identification string, "02". Accordingly, this first set of identical processes are assigned with the identification string "020". Had there been a second set of processes, they would have been assigned "021", and so on. The same steps as before apply to the new processes when they start traversing down the hierarchy. As mentioned before, identical processes can identify their relative positions with respect to their peers with the help of MPI's internal rank mechanism inside a communicator group [1]. Once again it should be mentioned that all of the previous issues are completely hidden from the user.

4.3.5 Obtaining Information about Peers

When identical processes are simultaneously spawned, they fall into MPI's same communicator group. Accordingly, they can figure out their identities, their relative positions inside the group, and any other relevant information using MPI's internal primitives (obviously all of these are hidden from the user). This is not the case when the processes spawned are not identical. The following discussion presents the scenario where non-identical processes are peers of one another and need to interact.

Once again, let us refer to the previous code. Since the **Generate**, **Geometry** and **Display** objects are created inside the context of **Root**, each of the child objects is passed with the information about its peers, with which it interacts. This information, in the form of references, is packed inside a **Params** object and is passed as a parameter during the construction of each of the handle objects (refer to the constructor of **Root**). For a default all-to-all interconnection topology, a handle is passed with the references of all the other peer-handles. A handle, in turn, passes this information to its associated "architectural skeleton" object while constructing it. Relevant pieces of the automatically generated code for **Generate** are re-inserted in the following:

```
class Generate : public SingletonSkeleton <PROT_Net>
{
public:

     class Params
     {
     public:
         HandleBase* h_73;
         HandleBase* h_113;
         Params(HandleBase* _h_73,HandleBase* _h_113) :
                                  h_73(_h_73),h_113(_h_113){};
         Params() :h_73(0),h_113(0){};
     };
     Params p_611;

     Generate(Params _p) : p_611(_p){};
     ...
     ...
}
```

In the previous code, when the **Generate** object is constructed by the associated handle, it is passed with the previous information as a parameter to one of its constructors. In that way, each of **Generate**, **geometry** and **Display** has references to the handles of its peers. The communication protocols use this internal information whenever peers need to interact with one another.

4.3.6 Process-Processor Mapping

When processes are mapped to processors, it is desired that processes that need to frequently communicate with one another be placed in a closer vicinity in the processor cluster. Many other factors, other than the physical distances of the processes, also arise, for instance: processor load, non-uniform network speed and bandwidth inside the cluster, network congestion at a specific time, and maintaining optimal load-balancing among processors. The topic of process-processor mapping, while taking into consideration all of the previous factors, is a complicated research issue in itself and the interested reader may refer to [30] for a discussion and links to various mapping related topics. The present implementation does not apply any specific mapping strategy and lets MPI handle this aspect (which applies a round-robin mapping scheme). Enough opportunity exists for exploring the mapping related issues inside many of the skeletons (for instance: the compositional and the data-parallel skeletons), and need to be researched in the future versions of this work.

4.4 Steps Involved in Building an Application

This section describes the steps involved in developing an application that uses the current textual specification language. It is assumed that the skeleton library and the applications reside inside a single directory, identified by the environment variable `SKELETON_HOME_DIR`. That directory is divided into several sub-directories: the `Lib` sub-directory contains the skeleton library, the `App` sub-directory contains the C++ applications, the `UI` sub-directory contains the applications written using the specification language, the `Include` sub-directory contains the various include

files, and the **Parser** sub-directory contains the specification language parser and associated files.

The following discussion presents the steps involved in building the **Jacobi** application. It is assumed that the application written using the specification language resides inside the single file **Jacobi.txt** (note that multiple files can also be supported) in the sub-directory **$(SKELETON_HOME_DIR)/UI/Jacobi**. A user issues the **make** command, which invokes the **Makefile** residing in the same sub-directory. This, in turn, invokes the parser, which automatically generates the C++ file **Pmain.cc** inside the sub-directory, **$(SKELETON_HOME_DIR)/App/Jacobi**. A sample **Makefile** might look as follows:

```
############################################################
# Makefile for application source.
# All rights reserved. 1998.
# Dhrubajyoti Goswami
############################################################

############################################################
# Macros
############################################################

PARSERDIR = $(SKELETON_HOME_DIR)/Parser
PARSE = parse
TARGET = Pmain.cc

############################################################
# Application source: user specifies this
############################################################

SRC = Jacobi.txt
APPDIR = $(SKELETON_HOME_DIR)/App/Jacobi

############################################################
# Rules:
############################################################

$(TARGET): true
        ln -s $(PARSERDIR)/*.pm .
        $(PARSE) -f $(SRC) -o $(TARGET)
        mkdir -e $(APPDIR)
        cp $(TARGET) $(APPDIR)

.DONE:
        rm -f $(TARGET) *.pm

true:
```

As the next step, the user changes directory to `$(SKELETON_HOME_DIR)/App/Jacobi`, where the automatically generated C++ source(s) resides. There the user invokes another **make** command to generate the executable. The corresponding `Makefile` is not shown here.

Finally, assuming that MPI is installed in the underlying cluster and is working properly, the corresponding executable file can be executed using the command: **run <executable_file_name>**. Here, **run** is a shell script that resides inside the sub-directory called `$(SKELETON_HOME_DIR)/Scripts`.

4.5 Summary

The chapter presents the key implementation aspects of the PASM model, starting with the textual user interface and its use in implementing various applications. The subsequent sections deal with the different implementation aspects of the model including the design of the skeleton library, its reusability and extensibility, the dynamic execution model, mechanisms for constructing the hierarchy, and finally the issue of process-processor mapping. The implementation of the PASM model is often abbreviated as the PASM system in the following chapters. The next chapter discusses the individual patterns in somewhat detail from the perspective of a "pattern language", a format that is presently widely adopted by the patterns community for writing about patterns in any discipline.

Chapter 5

A Pattern Language

Until now, parallel architectural skeletons have been discussed without bringing in the actual correlations between the skeletons and the various patterns in parallel computing. This chapter bridges that gap. Considering a pattern as a "problem/solution pair", a skeleton provides solution(s) for a pattern from the perspective of the model. The following discussion presents the inter-related solutions for patterns provided by the skeletons in the form of a pattern language.

5.1 Introduction

Since the visionary idea on patterns and pattern languages by the architect named Christopher Alexander [3], applied in the context of (physical) architectural design (e.g., buildings, bridges, hospitals, etc.), similar ideas are recently being widely adopted by the object oriented computing community and several other disciplines, some of which are not even related to computing. As mentioned in an earlier chapter, the "Pattern Languages of Program Design" series of books [37] are good

references for anyone interested in pattern-related topics covering a wide range of disciplines.

A *design pattern catalog* provides a set of individual, not-necessarily related solution techniques to common design problems. On the other hand, a *pattern language* provides a collection of interrelated solution techniques to common design problems in a specific problem domain. A pattern language is not a formal language, but rather a collection of interrelated solution techniques that together provide a vocabulary for talking about a specific problem or a collection of problems.

The skeletons discussed in this thesis are all woven together by the generic PASM model, which defines them, and these interrelated skeletons are used together to solve commonly occurring problems encountered in network-oriented parallel computing. Each skeleton is a physical manifestation of a pattern in parallel computing (e.g., the data-parallel skeleton is a semi-concrete physical manifestation of the data-parallel pattern), where each pattern is represented as a "problem/solution pair". Solutions for patterns realized by the skeletons are interrelated with each other in the context of the model, and together they form a pattern language that provides techniques for designing and implementing network-oriented parallel applications. Figure 5.1 diagrammatically illustrates the relationship between the various skeletons and the associated patterns in this pattern language.

The following discussion presents the current set of patterns that has been realized by the generic model, i.e., patterns and their solutions realized through the parallel architectural skeletons. The discussion uses the present commonly accepted format in pattern writing [49].

In describing each pattern, which is essentially a "problem/solution pair" for a commonly occurring problem, the problem is discussed in a general context which

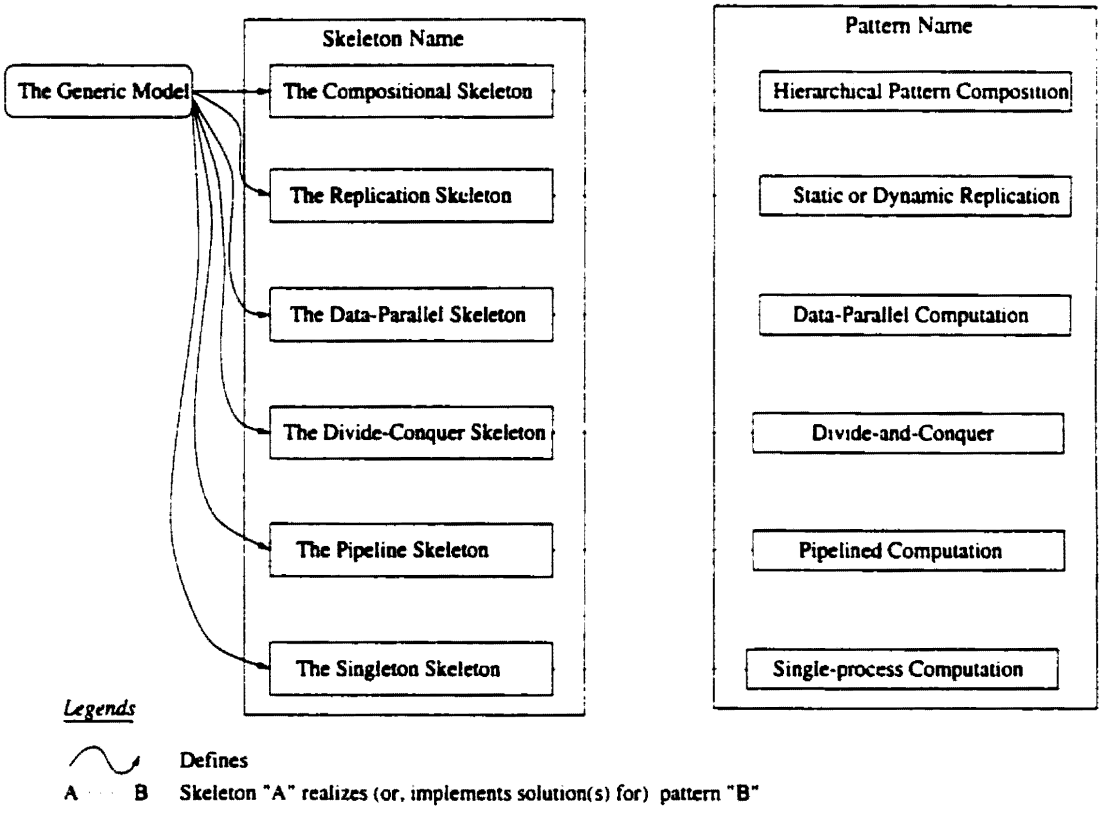


Figure 5.1: Relationship between skeletons and patterns in the language

universally applies to parallel computing. Solutions to the problem are often presented exclusively from the architectural skeleton perspective, meant for a user of the skeleton-library. The examples are illustrated only from the perspective of the architectural-skeleton approach.

Since there is no notion of a *process* in the architectural-skeleton approach, the following conventions in terminology are followed in the rest of the discussion. A *process* in the general context is equivalent to a *sequential module* (that is implemented as an extension of the *singleton skeleton*) in the architectural-skeleton context, and the two terms are used interchangeably. A *module* in the general context means a modular, single- or multi-process entity. A *module* in the architectural-skeleton context conveys its usual meaning.

Descriptions of the patterns follow next. Due to space constraints, two of the patterns are discussed for illustrative purposes in full detail (indicating what a future reference manual might look like). Discussion of the compositional and the data-parallel patterns are vast topics and their inclusion here would explode the size of this document. Accordingly, the rest of the existing patterns are briefly discussed, only partially conforming to the set standard in pattern writing.

5.2 Pattern: Dynamic Replication

Context: The following problem may arise in the general context of any parallel application development that uses the message passing paradigm, and involves single or multiple interacting modules.

The problem described below is applicable in either of the following situations: (a) a sequential module (which could be implemented using the *singleton skeleton* in

this language) has to interact with other modules in an application. The module has to perform repeated identical computations on different sets of data, for instance, in different iterations of a loop where the loop iterations are independent of each other. However, in comparison with the other modules that it interacts with, the workload of the sequential module is high enough to slow down the entire application. (b) A stand-alone sequential module has to repeatedly do the following: read in some data from an I/O device or a file, perform some computation on the data, and finally output the result to an I/O device or a file. However, the computation phase is a bottleneck.

Problem: How to increase the throughput of the out-of-pace module in the previous context, and thus possibly enhance overall performance, without modifying the rest of the application?

Forces:

- Achieving speedup with minimal possible modifications is the biggest consideration here.
- It is possible that multiple developers are involved, specially when the first situation in the previous context applies. In that case, a bottleneck in part of the application needs to be resolved without involving others.
- The first bottleneck-situation in the previous context frequently arises, for instance, when the interacting modules form a pipeline. Each pipeline stage performs some repeated identical computations inside a loop, where each iteration of the loop is independent of the other, and communication is involved exclusively at the beginning and the end of each iteration. One or more of the pipeline stages might participate in relatively more time-intensive computations as compared to the rest, which in turn pulls down the performance of

the entire pipeline (for instance: refer to the graphics animation application discussed in the previous chapter).

Solution: A solution to the problem is presented here from the architectural-skeleton perspective, intended for a user of the skeleton-library. In discussing the solution, it is assumed that the original sequential module has the following repetition code-structure:

```
SomeModule EXTENDS SingletonSkeleton
{
    Rep {
        ...
        while (some_condition == True) {
            Read_in (data); // ''data'' is read-in from some I/O device,
                            // file or from another module.
            Process (data); // Process data that is read-in above.
            Write_out (data); // Output processed data to some I/O device,
                              // file or another module.
            Revalidate_condition(some_condition, ...); // Re-validate
                           // ''some_condition'' to see if it still holds
        }
        ...
    }
    ...
}
```

A solution to the problem involves the following steps:

- **Step 1**: Identify the repeated identical computations performed by the sequential module, `SomeModule`, that can be replicated to run concurrently in order to possibly enhance the throughput of the module, provided multiple free processors are available, without affecting the rest of the application. For instance, in the previous code, it is the method `Process(data)` inside the `while` loop.

- **Step 2**: Enclose the repeated part of the computation into a separate sequential module (in the previous case, it is the method `Process(data)`). Here,

this sequential module is named as **Worker**, since it performs the pure computation part. A code segment for **Worker** is shown in the following.

- **Step 3:** Replace the sequential **SomeModule** with another module of identical name, but this time extending the *replication skeleton*. The **Worker** module, in step 2, becomes its child.

- **Step 4**: The external protocol of **SomeModule** remains the same as before, i.e., if it is at the root of the hierarchy, its external protocol is **Void**. Otherwise, the external protocol of **SomeModule** becomes the internal protocol of its parent.

- **Step 5**: The internal protocol of the replication module, **SomeModule**, is PROT_Repl. Accordingly, PROT_Repl becomes the external protocol of each replicated **Worker**. Use the communication-synchronization primitives inside PROT_Repl to restructure the code-segments of both the modules, as illustrated in the following self-explanatory code segments.

- **Step 6**: (Optional) Later replace the sequential **Worker** module with any other suitable module supported by this language, if deemed necessary.

The following code-segments illustrate one particular solution. However, other variations are also possible depending on the specifics of the problem.

```
SomeModule EXTENDS ReplicationSkeleton
{
    CHILDREN = Worker
    Rep {
       ...
       for (;;) {
           int  success = True;
           while ((some_condition == True) && (success == True)) {
                Read_in (data);
                success = SendWork(data); // Dynamically send work-load to a
                // free worker. If none is free, spawn one while performing a
                // suitable load-balancing strategy. It is a member primitive
                // of the internal  protocol, PROT_Repl, and is described in
                // the following.
```

```
                Revalidate_condition (some_condition, ...);
            }
            if (success == False) { // Unsuccessful in assigning work-load to a
                                    // worker. So, process it on your own.
                Process (data);
                Write_out (data);
            }
            CollectResults(); // Collect as many available results as possible.
                              // This procedure is defined in the following.
            if (some_condition == False) break;
        }
        while (ResultsPending()) CollectResults(); // Collect all remaining
            // results. ResultsPending() is a  member primitive of PROT_Repl.
    }
    LOCAL {
          void Collect_Results()
          {
              int success;
              while ((success = ReceiveResultNB(data)) == True) {
                   // The above is a member primitive of PROT_Repl, and is
                   // non-blocking. Collect as many of awaiting results as
                   // possible.

                   Write_out (data);
              }
          }
          ...
    }
}

// Each of the replicated worker.
Worker EXTENDS SingletonSkeleton
{
    Rep {
        ...
        ReceiveWork(data); // This blocking version of receive is a member
               // primitive of the external protocol, PROT_Repl, and is a
               // counterpart of SendWork(data), used previously.

        Process (data);
        SendResult(data); // A member primitive of the external protocol,
                          // PROT_Repl.
    }
    ...
}
```

The previous code-segments convey the basic idea, based on a generic repetition structure. Other variations of the code are possible, depending on the specific situation (for instance, a multi-threaded representative could also be designed providing similar functionalities). Also, refer to the examples that follow for concrete illustrations, and a following section that describes the primitives for a detailed look at

their functionalities.

Examples: For a concrete example, refer to the graphics animation application discussed in the previous chapter. There, the out-of-pace sequential `Display` module was replaced with another module of identical name, that supports dynamic replication. It should be noted that none of the other modules in the application was affected by this change. The interested reader might want to compare the previous generic code with the code for the graphics animation application. It will be observed that the Collect_Results() portion of the previous solution is missing from the refined `Display` module.

Selected Primitives: A selected set of primitives from PROT_Repl is discussed in the following:

int SendWork(Wrapper& workload) : The range of actions performed by this primitive is elaborated in the following pseudo-code.

```
0. result := success.
1. if any free child module is currently available
   then
     1.1. apply some suitable load-balancing strategy  // Another research issue
          to assign workload to one of the free modules.
   else
     1.2. If the current number of child modules is within the prescribed
          maximum limit, spawn one child module while applying the needed
          load-balancing strategy.
     1.3. If successful in spawning
          then
             1.3.1. Assign workload to that child.
          else
             1.3.2. result := failure.
2. return result.
```

int operator<< (ReplicationPort& port, Wrapper& workload): This is an operator-overloaded variation of the previous primitive.

int SendWork(Wrapper& workload, int context) : This is another variation of the previous primitive that uses explicit user-specified context.

void RecieveResult(Wrapper& workload) : The range of actions performed by this primitive is elaborated in the following pseudo-code.

```
1. if no child has completed yet
   then
      1.1 block.
2. // Wakeup here
   2.1. Collect result from the first available child.
   2.2. If the child did not terminate itself
        then
           2.2.1 Mark child as free. // To be used in SendWork(...)
3. Exit.
```

void operator>> (ReplicationPort& port, Wrapper& workload): This is an operator-overloaded variation of the previous primitive.

int ReceiveResultNB(Wrapper& workload) : The non-blocking version of the above. It returns 1 on success.

void RecieveResult(Wrapper& workload, int context) : Another variation that uses explicit user specified context.

int ResultsAwaiting() : This primitive is used for checking if computation result is currently available from any of the child modules, without actually reading in the results. It returns 1 if at least one of the child modules has its results awaiting.

int ResultsPending() : Check if any more results are pending. It returns 1 on success.

The following are the child-specific counterparts of the previous send and receive primitives: **void ReceiveWork (Wrapper& workload)**, **void SendResult (Wrapper& workload)**. Other possible child-specific variants are not listed.

5.3 Pattern: Parallel Divide and Conquer

Context: The following problem may arise in the context of any parallel ap-

plication development. Here, a (sequential) module needs to implement a divide and conquer algorithm. During the *divide* phase of the algorithm, the application-problem to be solved is recursively divided into smaller and smaller sub-problems until some base condition is reached; then the sub-problem is solved by some suitable base-case algorithm. During the *conquer* phase of the algorithm, the solution to the original application-problem is formed by combining the results from the smaller sub-problems using a conquer-phase algorithm.

Problem: How to implement a parallel version of divide and conquer?

Forces:

- The divide and conquer pattern is encountered in a large number of problems, ranging from searching (e.g., binary search), sorting (e.g., merge and quick sort), various graph algorithms (e.g., recursive graph partitioning, finding the closest pair of points in a graph), selection algorithm (i.e., to find the k^{th} smallest element in a list of n elements) to an optimal $O(n^{2.81})$ algorithm for the multiplication of two matrices, to name a few.

- Successive dividing of the problem into smaller sub-problems and the subsequent conquering results in a divide-and-conquer tree structure, which grows in size during the dividing phase and shrinks during the conquering phase. Data is always input at the root of this tree. Output need not always be at the root level (for instance, refer to the recursive partitioning of a graph). Leaves of the tree correspond to the base condition. Also refer to section 2.1.

- For a recursive (i.e., sequential) implementation, each non-root node of the previous divide-and-conquer tree correlates to a recursive subroutine call that implements the particular divide and conquer algorithm. Each leaf node correlates to the base condition of the algorithm.

- Obviously, in a parallel implementation, each recursive call needs to be replaced by a separate process/thread that executes the subroutine. However, creation of a new process/thread for each invocation can result in reduced efficiency, simply due to the fact that each parent node of the tree has to wait idle until results from all its children become available. Thus, commonsense says that efficiency can be enhanced by assigning the workload of one of the children to the parent process/thread, consequently keeping all processes/threads and involved processors busy during the dividing phase of the computation.

Solution: In discussing the solution, it is assumed that the following is the generic structure of the recursive divide and conquer algorithm executing at each node of the divide and conquer tree:

```
Procedure DivideConquer(dataIn: Input, dataOut: Output)

   step 1. if the base condition is met
           then
              step 1.1. ProcessData (dataIn, dataOut); // Application-specific
                     // base-case procedure to straightaway process input when
                     // the base condition is met.
              step 1.2. go to step 5.
   step 2. PartitionData (dataIn, Part1, Part2,...,Partk); // Application-
                        // specific procedure for partitioning data into k
                        // sub-parts when the base condition is not met.
   step 3. Execute the following k recursive calls:
              step 3.1. DivideConquer (Part1, Out1);
              step 3.2. DivideConquer (Part2, Out2);
              ...
              step 3.k. DivideConquer (Partk, Outk);
   step 4. Combine (Out1, Out2,..., Outk, dataOut); // Application-specific
                          // procedure that combines the results from the k
                          // recursive calls to produce the final output.
   step 5. return.
```

A general solution to the problem replaces each recursive call with a separate process/thread that executes the same algorithm. A process-based general solution that is applicable to an MIMD distributed-memory environment is as follows, where

the previous procedure is replaced with a module (i.e., an executable file in the general context). Each executing copy of the module maps to a different node of the divide and conquer tree.

```
Module DivideConquer
dataIn: Input
dataOut: Output

  step 1. if root node
          then Read (dataIn); // Read in data from an I/O device, file or
                              // some other module.
          else Receive (Parent, dataIn); // Else receive data from parent.
  step 2. if the base condition is met
          then
              step 2.1. ProcessData (dataIn, dataOut); // Application-specific
                     // base-case procedure to straightaway process input when
                     // the base condition is met.
              step 2.2. go to step 7.
  step 3. PartitionData (dataIn, Part1, Part2,...,Partk); // Application-
                        // specific procedure for partitioning data into k
                        // sub-parts when the base condition is not met.
  step 4. for i := 1 to k, do the following
              step 4.1. handle h_i = Spawn (DivideConquer,...); // Spawn a
                        // process that executes "DivideConquer" and retain
                        // the handle to the process for future reference.
              step 4.2. Send (h_i, Parti);
  step 5. for i := 1 to k, do the following
              Receive (h_i, Outi);
  step 6. Combine (Out1, Out2,..., Outk, dataOut); // Application-specific
                          // procedure that combines the results from the k
                          // modules to produce the final output.
  step 7. if root node
          then Write (dataOut); // Write out result to an I/O device, file
                                // or send it to some other module.
          else Send (Parent, dataOut); // Else send data to parent.
  step 8. exit.
```

A specialized solution from the architectural skeleton perspective involves the following steps. The solution assumes that the input is provided as an 1-D array of marshal-able objects.

- **Step 1**: The previous module in the general context is substituted with a module in the context of this approach, that extends the *DivideConquer skeleton*. As a property of this skeleton, a module extending it has copies of itself as its own children (also refer to chapter 3).

- **Step 2**: For the copy of the module that is at the root of the divide-and-conquer tree, if it is also at the root of the hierarchy then its external protocol is `Void`. Else, its external protocol is the internal protocol of the parent.

- **Step 3**: For each non-root module, its external protocol is the same as the internal protocol, i.e., PROT_DivideConquer.

- **Step 4**: Use the primitives inside PROT_DivideConquer to restructure the previous code as follows. It assumes that a marshal-able array of type ttt (where, ttt can be built-in type or user defined) is input to the module.

For convenience to the reader, the member primitives of PROT_DivideConquer used in the following code-segments are commented between two star (*) marks for easy identification. The primitives are described in a following section.

```
DivideConquerModule EXTENDS DivideConquerSkeleton
{
    Rep {
        if (IamTheRoot()){ //* It is a member primitive of the internal
                // protocol, PROT_DivideConquer, which lets a module
                // dynamically identify itself regarding whether it
                // is the root of the divide-conquer tree.*
            ...
            Attt dataIn, dataOut; // Attt can be one of Aint, Achar,
                // Afloat, Auser, or any other marshal-able array
                // of standard data type.

            Readin (dataIn); // Application-specific procedure that reads
                // in the input data (either from an I/O device, a file
                // or receive the input from another module, in case
                // this module is a part of a bigger application). It
                // is defined locally in the following.

            DivideConquerProcedure (dataIn, dataOut); // The generic
                // divide-and-conquer procedure is defined locally in
                // the following.

            Writeout (dataOut); // Application-specific procedure that
                // writes out the output data (either to an I/O device,
                // a file or send the output to another module, in case
                // this module is a part of a bigger application). It
                // is defined locally in the following.

            return;
        }
```

```
    // Else
    Attt dataIn, dataOut;
    ReceiveFromParent (dataIn); //* A member primitive of
              // PROT_DivideConquer*

    DivideConquerProcedure (dataIn, dataOut); // The generic
              // divide-and-conquer procedure is defined locally in
              // the following.

    SendToParent (dataOut); //* A member primitive of
              // PROT_DivideConquer*
}

LOCAL {

   void DivideConquerProcedure (Attt& dataIn, Attt& dataOut)
   {
      if (BaseConditionIsMet(dataIn)) // Check if the base condition
            // is satisfied. It is defined locally in the following.
      {
           ProcessData (dataIn, dataOut); // Process input data
              // straightaway if the base condition is satisfied. It
              // is defined locally in the following.
      }
      else {
         Aint PartitionInfo;
         CreatePartition (dataIn, PartitionInfo); // Create the
              // application-specific partitioning of the input
              // data. It is defined locally in the following.

         PartitionData (dataIn, PartitionInfo); //* A member primitive
              // of PROT_DivideConquer, used for dynamically creating
              // the children modules (copies of itself) and then
              // dividing the input among them, based on PartitionInfo.*

         Auser<Attt> Results;
         CollectResults (Results); //* A member primitive of
              // PROT_DivideConquer, used for collecting the results
              // from the children.*

         Combine (Results, dataOut); // Application specific procedure
              // for combining the results from the children. It is
              // defined locally in the following.
      }
   }

   // The following application specific methods need to be implemented
   // by the user. They can be defined either locally or globally.

   void Readin (Attt& dataIn) {...}

   void Writeout (Attt& dataOut) {...}

   int BaseConditionIsMet (const Attt& dataIn) {...}

   void ProcessData (Attt& dataIn, Attt& dataOut) {...}

   void CreatePartition (Attt& dataIn, Aint& PartitionInfo) {...}
```

```
        void Combine (Auser<Aint>& Results, Atttt& dataOut) {...}
    }
}
```

Examples: For a concrete example, refer to the previous chapter for an implementation of quick sort using the divide and conquer skeleton. The interested reader might want to compare the previous generic code-structure and the concrete example in the previous chapter to get an idea about the similarities and the differences. Notice that the differences arise only in the application-specific aspects. The application-independent aspects and the overall code-structure (or, code-skeleton) remain identical for both.

Selected Primitives: The following is a selected set of primitives from the protocol, PROT_DivideConquer. These primitives are applicable when the input is in the form of a marshal-able, 1-D array of objects (either user- or system-defined).

int IamTheRoot(): This primitive lets a module dynamically determine if it is the root of the divide and conquer tree. It returns 1 if that is the case.

void SetTreeWidth(int width): Dynamically set the width of the tree, which is by default two. In this way, it is possible to create a multi-width divide and conquer tree.

int PartitionData (MarshalableArray& dataIn, Aint& PartitionInfo): Create n copies of children (i.e., copies of itself), where n is the current width, and distribute the input "dataIn" to the children, as based on "PartitionInfo". The array "PartitionInfo" contains the lower index and the size of each data partition, and its length must be even and at least four. If the current width of the tree is n (either by default or set by `SetTreeWidth(...)`), then the length of "PartitionInfo" must be $2 * n$. It returns 1 on success.

void CollectResults (Auser<Attt>& Results): Collect results from all the n children, and return them in "Results". It blocks until all the results are available. `Auser<Attt>` is a user-defined marshal-able array of type `Attt`, where `Attt` is again a marshal-able array of user- or system defined type (e.g., Aint, Afloat, Achar, Adouble, Auser, etc.).

int CollectResultsNB (Auser<Attt>& Results): This is the non-blocking version of the previous primitive. It returns 1 on success.

The following are the other primitives that do not need further explanation: **void SendToParent (Attt& data), void ReceiveFromParent (Attt& data), int ReceiveFromParentNB (Attt& data)**.

5.4 Pattern: Data-parallel computation

Data-parallelism is one of the most frequently used patterns in parallel computing, applicable to a wide variety of applications starting from image processing to sparse system solvers to various sorting and searching algorithms to applications in neural networks, to name a few. As the name implies, here the parallelism lies in the data, i.e., a group of identical modules perform the same operations but on different sets of data. The identical modules can form different topologies (e.g., N-dimensional mesh, N-D hypercube). From the perspective of this model, the *data-parallel skeleton* implements the data-parallel pattern. The identical modules that perform the actual computations are the children of a data-parallel module that extends the data-parallel skeleton, and constitute its back-end. (Also refer to the replication skeleton discussed in this chapter. However, the differences are that the identical children of a replication module can dynamically grow and shrink in number, i.e., they do not have a fixed topology, and there is no interaction among peers).

The internal protocols of the data-parallel skeleton (e.g., PROT_nDMesh, PROT_Hypercube) are designed for specific topologies. Primitive operations inside these protocols can be broadly classified into several categories: collective operations (e.g., gather, scatter, reduce, barrier synchronization, etc.), topology specific point-to-point operations (e.g., nearest neighbor communication, selective communication), topology independent operations (e.g., broadcast). Each category contains an exhaustive range of primitives suitable for specific needs.

For an illustration, refer to the implementation of the Jacobi iterative scheme in the previous chapter.

5.5 Pattern: Hierarchical Composition

Control-parallelism is another frequently used pattern in parallel computing. Unlike data-parallelism, where a group of identical modules execute on different data sets, here a group of different modules (i.e., modules that execute different instructions) act on the same or different data sets. A typical parallel application is a combination of data- and control-parallelism.

The pattern named *Hierarchical Composition* provides solution techniques for arbitrarily composing an arbitrary number of modules. From the perspective of PASM, the *compositional skeleton* implements this pattern. The child modules inside the back-end of a compositional module (i.e., a module that extends the compositional skeleton) can be all identical, thus resulting in purely data-parallel computation. Alternately, the child modules may be all different, thus resulting in purely control-parallel computation. As another alternative, some of the the modules in the back-end may be identical and the rest may be different, thus resulting in a combination of control- and data-parallelism.

Each module in the back-end can include other modules as well (for instance: a compositional module can include other compositional module(s)). This, in fact, shows the ability to form a hierarchy with an arbitrary composition of modules. Hence, is the name *hierarchical composition.*

One of the main purposes of the compositional skeleton is to provide the needed flexibility to a user in application development. The internal protocol of the compositional skeleton is PROT_Net, which is intended to provide an MPI- or PVM-like message-passing parallel programming environment to the user. An MPI-like parallel programming environment can be supported inside the compositional skeleton, simply by replacing MPI-processes with PASM-modules and by supporting MPI-like message passing primitives inside PROT_Net. Many of these primitives have already been implemented. This facility enables a user to develop an application from scratch if deemed necessary, or to intermix already supported patterns with arbitrary composition if an application demands so.

For an illustration, refer to the graphics animation application in the previous chapter, and its subsequent hierarchical refinement.

5.6 Pattern: Pipeline

Pipeline is a special form of control-parallelism, and the compositional skeleton described previously can be used for constructing a pipeline. However, pipeline itself is a frequently used pattern in parallel computing, and accordingly the *pipeline skeleton* is specifically designed for this purpose. Each stage of a pipeline is a parallel computing module that extends a specific skeleton. For constructing an N-stage pipeline, each stage is represented by a module and all the N modules can constitute the back-end of a pipeline module that extends the pipeline skeleton. Alternately,

for $N > 2$, the first stage or the last stage or both of them could merge with the representative of a pipeline module, and the remaining modules constitute its back-end. Data flow inside a pipeline can be uni- or bi-directional. Primitives inside the protocol, PROT_Pipeline, capture the various operations needed inside a pipeline.

When someone thinks of a pipeline, what naturally comes to mind is a single dimensional structure. However, there may be multi-dimensional pipeline-like patterns as well. For instance, one could think about the systolic array pattern, where the computation in each module (which is often called a *cell* in the context of a systolic array) is propagated to neighboring modules in a certain rhythmic fashion. Though presently the *systolic skeleton* is not designed and implemented, it may be considered in the future evolutions of the work if deemed necessary.

5.7 Pattern: Single process computation

In the general context, a parallel application is a composition of one or more interacting processes, where each process could be single- or multi-threaded. However, there is no notion of a process in this model. The singleton skeleton provides all the functionalities of a process from the conventional parallel programming perspective. Since the back-end of a singleton module is empty, its internal protocol P_{Int} is also void. Examples in this and the previous chapter have demonstrated usages of the singleton skeleton.

5.8 Conclusion

The chapter contains detailed presentations of two of the patterns of the pattern language that provides techniques for designing and implementing network-oriented

parallel applications. The rest of the patterns are discussed briefly. The patterns in the language are all woven together by the generic PASM model. The next two chapters discuss the performance issues, and the various software engineering related and other aspects of the model and the system.

Chapter 6

Performance Evaluation

The main purposes of this chapter are to demonstrate the following: (1) the PASM system has been efficiently implemented. (2) The performance of the system is comparable with MPI. (3) A suitable application of proper granularity should exhibit reasonable performance gain when implemented using the system.

Experiments were conducted to assess the performance of the PASM system. The results were compared with direct MPI-based implementations. The performance difference with MPI lies within 5%, which can be attributed to the fact that the skeleton-library is implemented as an extremely thin layer on top of MPI. The thin implementation layer generally implies that any application that demonstrates good performance with direct MPI-based implementation should provide similar performances with a skeleton-based implementation, under all identical conditions.

The following discussion presents the performance results, which can be broadly divided into two categories: application specific evaluation and application independent evaluation. The application-specific category involves results for some well-known parallel applications. Some of these results demonstrate the effect of

granularity on performance. These results can be further subdivided into two categories: performance based on timing and on software quality. The software engineering related aspects of the PASM model and the system are further elaborated in the next chapter.

The application-independent category compares the performances of certain primitive commands with direct MPI-based primitives, besides other application-independent performance measures to be discussed shortly.

6.1 Application Specific Evaluation

6.1.1 PQSRS

Parallel Quick Sort using Regular Sampling, abbreviated PQSRS [55], is a parallel version of quick sort, shown to be effective for a wide variety of MIMD architectures. It uses a combination of master-slave and 1-D mesh patterns, which is easily realized using the data-parallel skeleton for mesh topology and the singleton skeleton. The algorithm works in the following steps: (1) the master module partitions the data items to be sorted to the N children (i.e., slaves). Each child then performs sequential quick sort on its own data items, selects N data items as regular samples, and sends them back to the parent (i.e., master). (2) the master gathers the regular samples from all its children, sorts them, gathers $N-1$ pivot values and broadcasts them to the children. Each child partitions its portion of sorted items into N disjoint partitions, based on the $N-1$ pivot values. (3) Child i keeps the i^{th} partition and sends the j^{th} partition to its j^{th} peer. Thus, at this phase, each child has to communicate with all its $N-1$ peers. (4) Each child receives $N-1$ partitions from its peers, merges them with its own partition to form a single sorted

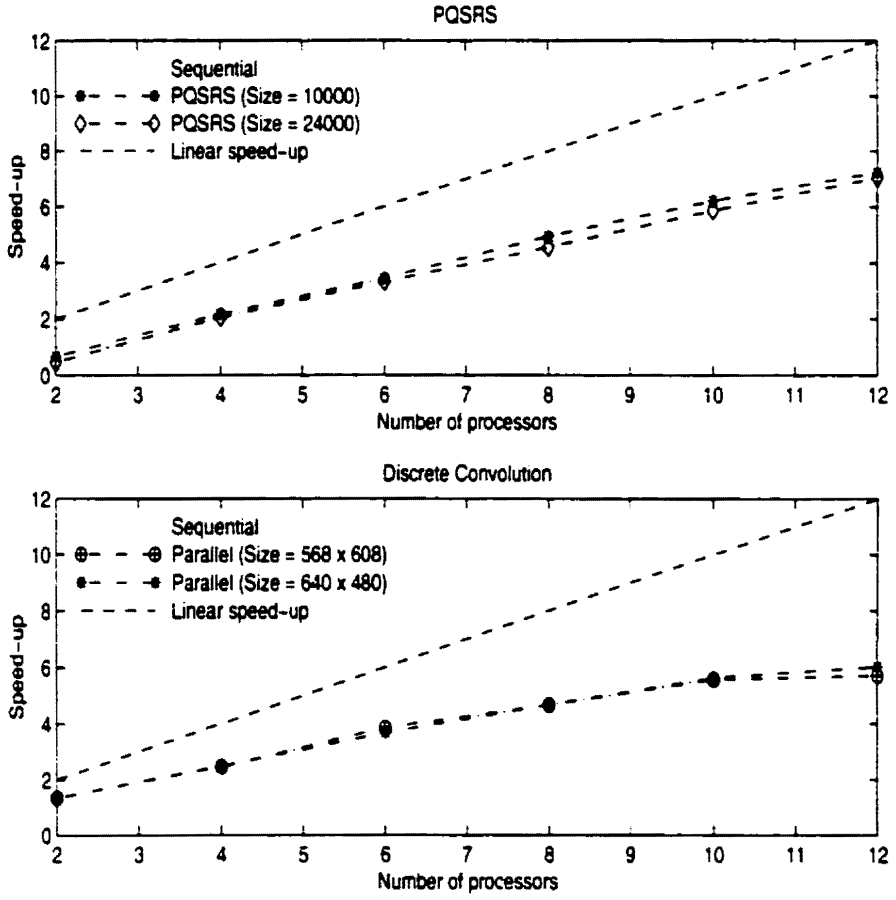


Figure 6.1: Speed-up ratio versus number of processors

list, and sends the sorted list back to the master. Finally, the master concatenates the sorted sub-lists from all its children to form the final sorted list.

PQSRS is a non-trivial algorithm which requires a considerable amount of peer-to-peer interaction among the slaves, which is supported by the internal protocol(s) of the data-parallel skeleton. It cannot be implemented using most other pattern based systems discussed previously. Figure 6.1 illustrates the results obtained for sorting 10000 and 24000 randomly generated objects using PQSRS. Time to compare two objects is approximately 0.2 ms. The underlying hardware is a cluster of

Sun Sparc workstations (each is an Ultra 10 Elite with 256 MB of RAM) connected by a 10-megabit Ethernet network. The speed-up ratio is measured with respect to the same sequential quick-sort routine used inside PQSRS. The performance difference with MPI is negligible and hence is not illustrated separately.

6.1.2 2-D Discrete Convolution

This is an image processing algorithm used for convoluting a given image through the application of a mask. The mask is applied to each image pixel to produce the convoluted image [51]. As compared to the previous application, this one is relatively simple. Like most other image processing algorithms, it follows the master-slave pattern where the slaves need not interact with one another.

Figure 6.1 illustrates the results obtained for the parallel discrete convolution of two pixel images of sizes 640 × 480 and 568 × 608. The mask used in each case is of size 10 × 10. The underlying hardware is identical to that used in the previous application. The speed-up ratio is measured with respect to the best sequential algorithm. The experiments demonstrate almost identical performances with both the images, however there is observed performance degradation with a 5×5 mask in each case (not shown in the figure) due to non-optimal computational granularity. More on the effect of granularity on performance is discussed in the next experiment.

6.1.3 Jacobi

The Jacobi method and its implementations have already been discussed in chapters 4 and 5. As compared to the previous two applications, it is a relatively finer-grain

application (e.g., four floating point additions and one division per node of a graph, as compared to the sliding of an entire mask over each image pixel in the case of the discrete convolution algorithm). As the code fragment illustrates, the parallel implementation requires nearest-neighbor communication. Compare it with the PQSRS algorithm, where each computing element (i.e., slave) has to communicate with all its peers.

The granularity (i.e., the ratio of computational time to communication overhead, between two successive communication points) can be increased by mapping more nodes per processor, provided that the corresponding rate of increase in communication overhead is less than the rate of increase in the number of nodes per processor. If granularity is more than a certain optimal value, which can vary from situation to situation, theoretically there should be speed-up. Otherwise, the parallel application shows performance degradation.

Figure 6.2 illustrates the effect of granularity on speed-up in the case of Jacobi, for multiple-sized square grids. As it is observed, if the granularity is too small (e.g., in the case of a 50×50 grid), there is visible slow-down. The optimal speed-up is observed for a grid size close to 500×500. There are almost identical speed-ups for grids of dimensions 100, 1000 and 2000. When the grid becomes too large (e.g., size 2500×2500), there are other overheads due to message fragmentation and swapping of memory space, etc., which might have contributed to the reduction in speed-up.

For a given grid size, division of the grid into n processors should ideally give a speed-up of n. However, in practice, this is not the case due to the communication-related and other overheads. As the number of processors increases, the computational granularity per processor decreases. Beyond a certain threshold, there is no added benefit in further dividing the grid into smaller sub-pieces. That threshold is not observed in the previous figure due to the lack of sufficient number of pro-

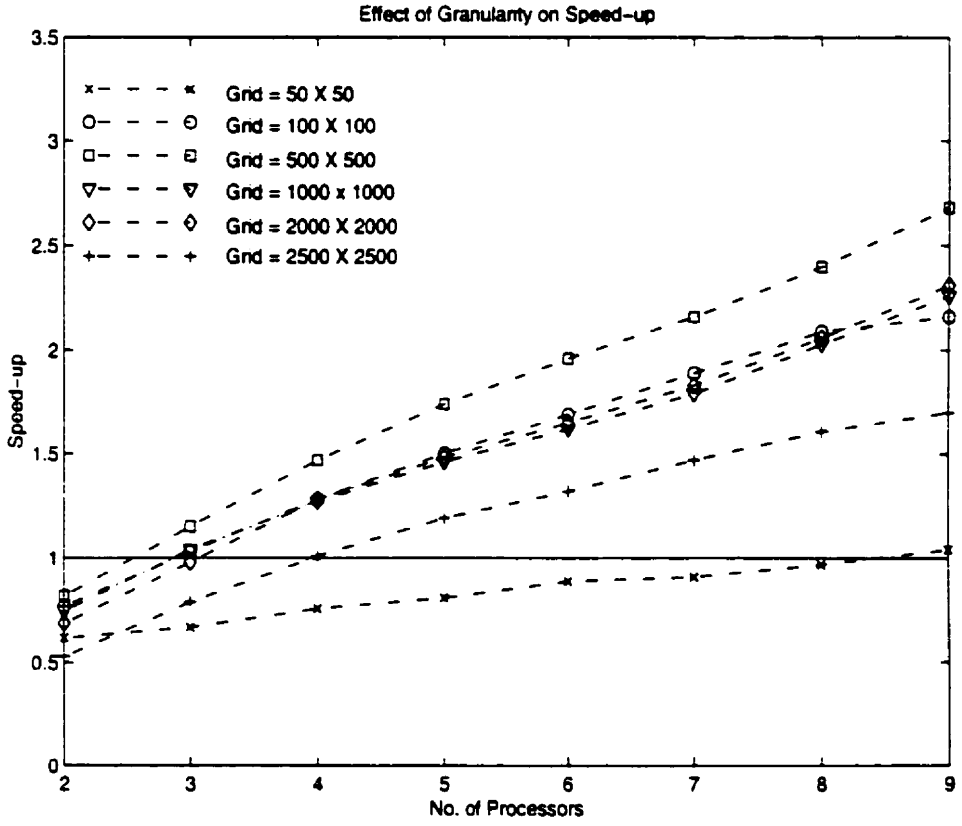


Figure 6.2: Effect of granularity on speed-up

cessors while conducting the experiments. The underlying hardware in this case is identical to that used in the previous two experiments.

6.1.4 Software Quality Measurement

As part of the Master's research of another graduate student, a comprehensive study was conducted to assess the software quality related aspects of PASM and some other related systems. The concept of software metrics is well established and a variety of software metrics have been used over time to measure the qualities of software products.

In this study, some candidate metrics for measuring software qualities, especially complexity, were collected (e.g., Halstead software science metrics [36], McCabe's cyclomatic complexity metrics [48]). The experiments involved PASM, Frameworks [62], Enterprise [56] and direct implementations using MPI. The study suggests that the use of architectural skeletons significantly lowers software complexity as compared to the code written from scratch using MPI. A detailed discussion of the study is beyond the scope of this thesis. The interested reader can refer to the comprehensive description of the work [68, 69].

6.2 Application Independent Evaluation

The measurements in this section can be subdivided into two categories: (1) comparison of certain primitive operations with equivalent MPI-based primitives, and (2) application-independent evaluation of certain patterns implemented using architectural skeletons, in comparison with equivalent sequential implementations.

As mentioned before, the performance difference with MPI is found to be rather insignificant, and hence, the first category of measurements confines only to the first of the following set of measurements, merely for the sake of completeness. The rest of the measurements fall into the second category.

6.2.1 Comparison of Some Basic Primitives with MPI

The `Send` and `Receive` primitives inside PROT_Net were compared with their counterparts, `MPI_Send` and `MPI_Receive`, inside MPI. Since the mentioned primitives inside PROT_Net (as well as other similar primitives inside PROT_Net and the other protocols) perform automatic data marshaling and un-marshaling, it makes sense to include the equivalent message-packing and unpacking times while measuring the MPI-based times. The table on the next page illustrates the best measured times for sending and receiving 1000 messages with varying message sizes, and the percentage differences. For each message size, the best time is taken out of at least five different runs under all identical configurations.

As the results suggest, the performance differences are insignificant and the varying network load conditions are the main contributing factors for the fluctuations in the results, which cannot be explained otherwise.

Message Size (Integer)	Send (PROT_Net) Best time (s/1000 msg)	MPI_Pack + MPI_Send Best time (s/1000 msg)	% Difference
20000	11.42	11.45	-0.26%
40000	24.524	24.232	1.21%
80000	50.09	49.56	1.07%
120000	61.74	62.87	-1.79%
Message Size (Integer)	**Receive (PROT_Net) Best time (s/1000 msg)**	**MPI_Receive + MPI_Unpack Best time (s/1000 msg)**	**% Difference**
20000	11.03	11.07	-0.36%
40000	22.39	22.43	-0.18%
80000	43.79	42.62	2.75%
120000	60.37	60.56	-0.31%

As another measure of comparison, the amount of code to be written by the user using both the approaches are also compared. In the previous test case, the minimal size of code to be written using the skeleton-based approach is approximately 1400 bytes (56 lines with all comments and blank lines removed). The same functionality could be achieved using MPI with a total user-written code size of approximately 3400 bytes, spread over two files (105 lines altogether with all comments and blank lines removed). The reduction in user-written code size is approximately 59%, and the reduction in the number of lines is approximately 47%. Note that byte size comparison will also depend on the lengths of the variable- and procedure-names used.

Other similar measures for the application-specific category were included inside software metrics measurements, discussed in the previous section.

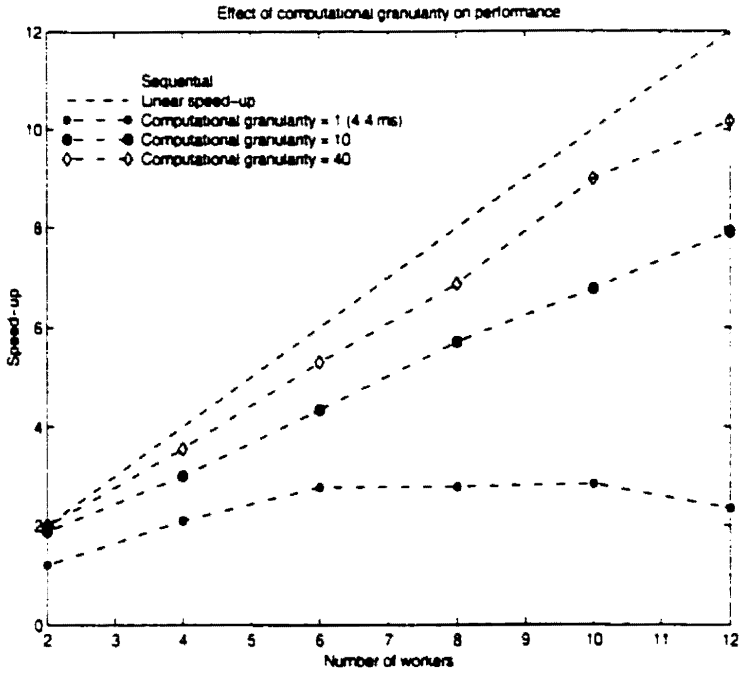


Figure 6.3: Effect of granularity on performance

6.2.2 Effect of Granularity on Master-Worker Performance

The next set of experiments involves application-independent performance evaluation of certain patterns, implemented using the architectural skeleton approach. The first in this category involves the master-worker pattern, implemented using the replication skeleton. The experiment involves a fixed amount of workload, distributed among replicated workers. The maximum number of workers concurrently present is varied. One worker is mapped per available processor.

The experiment is carried out with three levels of computational granularities per worker: granularity 1 (approximately 4.4 ms), granularity 10 and granularity 40. Granularity is introduced via 5 floating point multiplications inside a loop. With a computational granularity of 1, there are altogether 4800 calls to workers,

and a total of 9600 messages (send and receive) of fixed message size. With a computational granularity of 10, the total number of calls to workers (and also the number of messages) is reduced proportionately to 480, while the message size remains the same as before. Same is the case with the granularity of 40, where the number of calls is reduced to 120. Figure 6.3 illustrates the results obtained with different numbers of workers present at a particular run.

As the figure shows, higher computational granularity gives near-linear performances. On the other hand, with lower computational granularity (of one), the performance is sub-linear with a higher number of workers. This phenomenon can be attributed to the fact that communication cost (associated with 9600 messages) and the maintenance cost of the workers dominate the computational time on each worker. As a result, performance gradually degrades with a higher number of workers. On the contrary, communication cost is much reduced with a higher computational granularity (for instance, a total of 240 messages are associated with a granularity of 40), and also the maintenance cost of the workers becomes negligible as compared to the computational granularity per worker. These factors contribute towards the near linear performances with higher granularities.

6.2.3 Pipeline with and without Replication

The graphics animation application and its subsequent refinement is discussed in detail in chapter 4. This set of experiments involves a similar situation with three modules: `Producer`, `Worker` and `Consumer`, which have different levels of computational granularities and together they form a pipeline. Once the pipeline becomes full in a parallel run, each of the modules is working concurrently and hence the overall speed-up is governed by the slowest of the modules. For instance: assuming

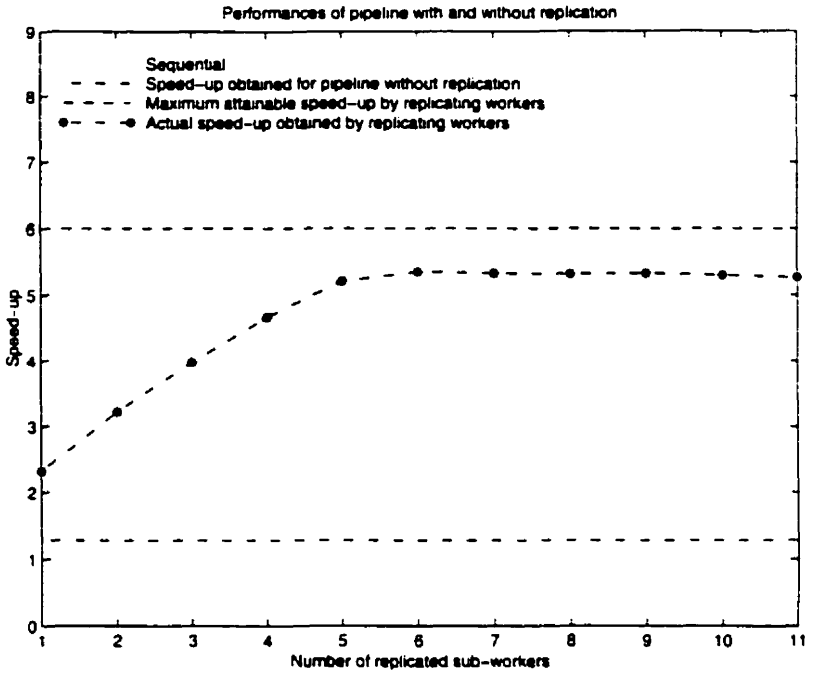


Figure 6.4: Performances of pipeline with and without replication

that the granularities of the modules are 1, 9 and 2 respectively, a sequential run has a granularity close to 12 per iteration. On the other hand, a parallel run (once the pipeline becomes full) has a granularity close to 9, ignoring the communication overhead, and thus the maximum attainable speed-up is close to $12/9 = 1.33$. This can be improved by replicating the workload of the slow **Worker** among subordinate workers, i.e., **SubWorkers**. Theoretically, with 9 **SubWorkers** present concurrently, one of them is producing a result every 1 time unit, and hence, the computational granularity of the **Worker** should reduce approximately to 1. The **Consumer** module, with a granularity of 2, now dominates the pipeline and the maximum attainable speed-up is close to $12/2 = 6$. This can once again be improved by replicating the **Consumer**, which can theoretically provide a maximum speed-up of 12.

This set of experiments involves the exact situation with the aforementioned

granularities of the three modules. Computational granularity of one corresponds to approximately 5 ms. Only the `Worker` module is replicated, and hence the maximum attainable speed-up is 6. Figure 6.4 illustrates the results obtained with and without replication. As shown in the figure, the maximum attained speed-up without replication is 1.28, while the theoretical limit is 1.33.

With replication, a saturation point is reached with approximately 6 sub-workers. This can be attributed to the fact that, starting at the saturation point, the maintenance cost of the sub-workers and the communication overhead (between worker and sub-workers) start dominating over the computational granularity of each sub-worker, thus offsetting any benefit henceforth.

The experiments were carried out at different times of the day, spread over two days, and the best readings are taken. Moreover, due to the brief vacation period at the end of the term, the system and the network load was quite minimal during the times of the tests.

6.2.4 Performance of Pipeline with Varying Granularity

The next set of experiments involving pipeline investigates the effect of granularity on performance. There are several variable factors that need to be considered: total number of pipeline stages, computational granularity per stage assuming uniform granularity across all stages, and message size. For this set of experiments, the total number of pipeline stages is fixed at 8 where all stages have equal granularity. Granularity is directly proportional to the ratio of the computational time between two successive communication points (that is, the computational granularity) to the communication overhead. Theoretically, higher granularity gives better performance, as long as granularity is below some threshold value that depends on many

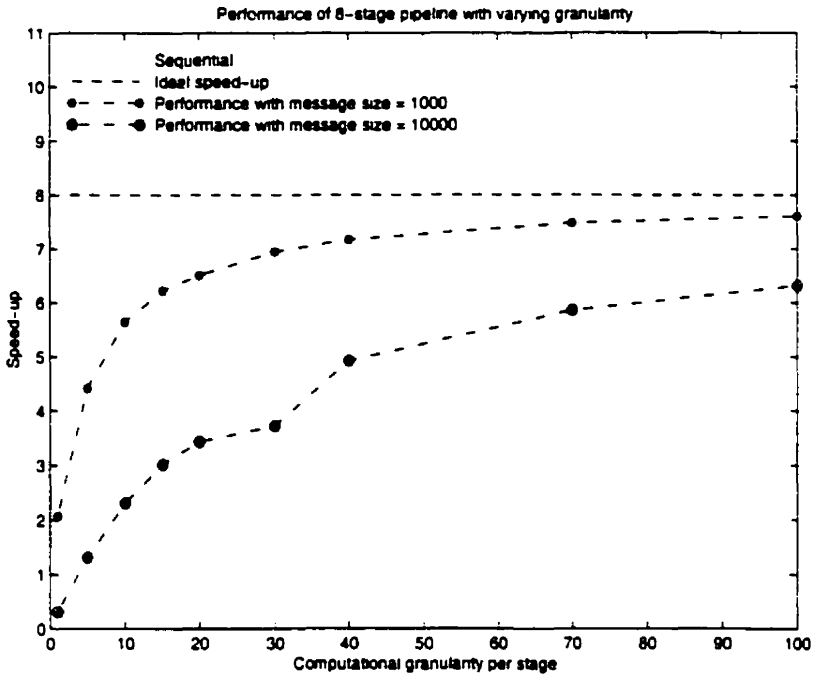


Figure 6.5: Performances of pipeline with varying degrees of granularity

physical factors of the underlying architecture.

With 8 pipeline stages, each of equal granularity, the maximum attainable speed-up is 8. Figure 6.5 illustrates the results obtained with varying degrees of granularities per stage. Throughout the experiments, each pipeline stage is mapped per available processor. The computational granularity of 1 corresponds to approximately 0.5 ms. With fixed message size, higher computational granularity corresponds to higher granularity per stage. On the other hand, with fixed computational granularity, higher message size corresponds to lower granularity per stage. The graphs demonstrate better performance with higher granularity, thus conforming to usual predictions.

6.2.5 Conclusion

The chapter presents the application-specific and application-independent performances of the PASM system. As the comparisons with MPI suggest, performance differences with direct MPI-based implementations are negligible, which is due to the fact that the skeleton-library is implemented as an extremely thin layer on top of MPI. The software engineering related aspects of PASM are discussed in the next chapter.

Chapter 7

Crucial Issues, Future Directions

This chapter focuses on several important aspects of the PASM model and the associated system. These include the fundamental contributions of the research, various software engineering related issues of the model and the system, and its comparison with some other related works. The following section on future research directions emphasizes some of the core issues that need to be considered in future versions of this work.

7.1 Fundamental Contributions

The most fundamental contribution of the parallel architectural skeleton model is its genericness, which leads to the other contributions of the approach (e.g., flexibility and extensibility). Some of the other fundamental contributions are: modularity (which contributes towards its object-oriented design and implementation), the capability of describing a skeleton independent of other skeletons (which contributes towards extensibility), and the capability of implementing the model in

C++ without requiring any language extension. Some of these issues are further discussed in the following.

7.2 Software Engineering Issues

The following discussion presents the various software engineering related aspects of the model and the system. Some of these issues were previously discussed in the book-chapter by this author [34].

7.2.1 Reuse

There are two types of reuse that can be mentioned: (a) reuse of code for patterns, and (b) reuse of application code. The first type of reuse is quite evident in this system, since each architectural skeleton extracts and implements the structural and the behavioral attributes associated with a pattern in an application-independent manner. The various parameters associated with these attributes (for instance: dimensions of a mesh, width of a divide-conquer tree, selection of appropriate protocol(s), etc.) enable the same skeleton to be configured to the needs of different applications as abstract parallel computing modules. The abstract modules become concrete with the insertion of application code.

Regarding the reuse of application code, a parallel application can be viewed as a restructuring of the original sequential code with embedded parallelism constructs. A smart restructuring enables good portions of the original sequential code to be reused. For instance, in the graphics animation example in chapter 4, the procedures *DoHidden*(...), *DoConversion*(...) and *WriteImage*(...) are reused from the original sequential code, except for minor changes related to the parameter type(s).

Moreover, these reused procedures contain the majority of the code for the entire application.

7.2.2 Genericness

As opposed to being ad hoc, each architectural skeleton is defined in a generic fashion (that is, in a manner independent of any pattern or application) with its canonical set of attributes. Many useful patterns in parallel computing are realized inside the frameworks of the generic model (refer to Figure 4.4). Each parallel computing module can interact with other modules via standard interfaces (i.e., the representatives), a well-defined set of protocols and using a universal set of rules. The generic approach enhances usability.

7.2.3 Flexibility

Flexibility is one of the major concerns associated with all pattern based approaches [63]. Often, if a certain desired pattern is not supported by a pattern-based system, there is no alternative but to abandon the idea of using the particular approach altogether.

MPI [35] is known to be extremely flexible because of its proven applicability in solving a vast majority of parallel applications. Often different solution strategies can be planned out while solving an application using MPI, which gives the user complete flexibility. Inside the frameworks of PASM, that type of flexibility can be achieved if the features of MPI can be directly supported. This is the main idea behind the compositional skeleton and its associated protocol, PROT_Net. The compositional skeleton in conjunction with its internal protocol PROT_Net

is intended to provide an MPI-like parallel programming environment within the frameworks of the model, and it can be used to substitute patterns if an application demands so.

Moreover, a compositional module is like any other module from PASM's perspective. Consequently it can be used in conjunction with the other patterns supported by the model. This type of uniformity should provide added flexibility to the user.

7.2.4 Extensibility

As mentioned previously, lack of extensibility is another major concern associated with most pattern-based approaches [63]. Most of these systems are hard-coded with a limited and fixed set of patterns, and often there is no clear way to add new patterns to the system when need arises.

From the implementor's or an experienced user's perspective, certain features of the object-oriented design, in conjunction with the generic nature of the model, favor reuse and extension of the skeleton library. The generic model helps, because it provides a clear picture regarding the different components of a skeleton and their functionalities (compare it with a totally ad hoc approach). Furthermore, from PASM's perspective, each module is an independent entity whose only interface with the outside world is through its representative and the adaptable external protocol. Accordingly, what the outside world sees of a module are its actions (i.e., input/output and any observable side effects), without knowing exactly how these actions are carried out internally. In other words, each module is an independent self-contained entity that acts as a black-box to the outside world. For the same reasons, each module can be designed independent of others. In other words, the

PASM model inherently supports extensibility.

From the PASM system's perspective, many of the object-oriented features that are supported in C++, for instance: polymorphism (through the use of C++ templates, inheritance and overloading), favor the reuse and extension of the existing skeleton library. New classes can be defined by extending the existing ones, thus enabling the design and addition of new skeletons and protocols with added functionalities. Completely new skeletons and protocols can be designed by extending the base classes (refer to Figure 4.4). In each case, a collection of pre-existing virtual methods need to be over-written and new additional methods might need to be designed in order to reflect the characteristics of the newly designed skeleton.

7.2.5 Hierarchical Development and Refinement

A parallel computing module can contain other modules, and hence, application development using PASM is distinctly hierarchical. Moreover, a parallel computing module can be viewed as a black-box, where the only visibility from the outside world is in the action of the module, and in its interface and interaction with other modules. As long as these factors remain unchanged, the module can always be replaced with another module, which implements some other pattern(s), without affecting the rest of the application. Such type of replacement for betterment is called a refinement. The hierarchical model leads to hierarchical refinement.

Hierarchical refinement is illustrated for the graphics animation example in chapter 4, where the singleton `Display` module is refined to a dynamically-replicated module of identical name. Figure 4.7 illustrates the affect of refinement on the hierarchy.

7.2.6 Separation of Concerns

Also known as *separation of specifications*, separation-of-concern is a desirable characteristic of all pattern-based approaches. Through the extraction of the application-independent components of patterns into architectural skeletons, there exists a clear separation between application code and application-independent issues. The application-independent components hide most of the low-level details related to process/thread creation and management, process-processor mapping, communication and synchronization, load balancing, data marshaling and un-marshaling, and architecture- and network-specific low-level details. These pre-packaged components are tested to be reliable, provided they are used correctly.

Application development using parallel architectural skeletons is clearly a multi-stage process (refer to Figure 3.1), where each stage is distinct from the others. The first stage provides pure application-independent abstractions. The clear separation of the low-level details allows a user to concentrate more on the application-specific issues.

7.2.7 Composition Using Patterns

A parallel computing module can contain other modules inside its back end, and thus, pattern-composition is an inherent property of the model. The *compositional skeleton* supports arbitrary composition of patterns inside its back end, with no restriction on the types of patterns that can be composed (refer to chapter 5). Thus, a compositional module, which is an extension of the compositional skeleton, can contain other compositional modules as well. Standard interfaces for all modules and a well-defined adaptation rule make pattern-composition extremely feasible.

7.3 Comparison with Related Work

The generic nature of the architectural-skeleton model is one of the features that distinguishes this research from other pattern-based approaches in parallel computing. As has already been discussed, the generic model contributes towards both flexibility and extensibility, which are some of the essential features lacking in most of the existing pattern-based approaches to parallel computing.

In the past, several parallel programming systems have supported frequently used parallel interactions [11, 12, 56, 61]. However, in all these cases a fixed number of high-level parallel interactions are hard-coded into the system. As a consequence, if a user's desired high-level interaction is not supported by a particular system, then the user has to adopt a different approach. To achieve higher flexibility, traditionally parallel programmers have relied on low-level communication libraries such as PVM and MPI. So there is clearly a trade off between the ease of development provided by the higher-level systems and the flexibility offered by low level primitive libraries. As discussed, in PASM, a user can mix high-level architectural skeletons with low-level MPI-like message passing (for instance, the internal protocol PROT_Net inside the compositional skeleton). Moreover, in all of previous systems, the supported patterns are tightly integrated into the implementation of the system, so there is no easy way of adding newer patterns without major modifications to the entire system. In PASM, on the other hand, every architectural skeleton is independent of other patterns, and thus, adding new architectural skeletons is a simple matter of extending the library of architectural skeletons.

Tracs [6] is one of the earlier systems that addresses the issue of extensibility. It is a graphical development system, where application development consists of two distinct phases: the definition phase and the configuration phase. During the

definition phase, the user graphically defines the three basic components of an application: the message model, the task model and the architecture model. The architecture model defines the software architecture of the parallel application in terms of message and task models. An architecture model defined during this phase can be saved in a user-defined library for later use. During the configuration phase, the programmer constructs the complete application from the basic components, either defined during the definition phase or selected from the system libraries or both. Evidently, *Tracs* supports the idea of extensibility by providing support for an extensible library of user-defined architecture models. However, the type of extensibility realized by *Tracs* is restrictive. For instance, in *Tracs*, a user can graphically create a 5-slave master-slave pattern and save it inside the library for future use. However, a generic master-slave pattern is more useful for this purpose.

As far as known to us, *DPnDP* [66] is the first system that addresses both the issues of flexibility and extensibility. It was a nice attempt, but unfortunately it concentrates only on the structural aspects of a pattern and ignores the behavioral aspects altogether (for instance: parallel computing model, communication-synchronization behavior inside a pattern). In spite of its limitations, *DPnDP* was a good learning experience and it set up the initial stage for this research.

There are various other systems and research projects in the object-oriented domain that are intended to facilitate parallel application development. A majority of them are based on C++ or its extensions. Some of these systems are pattern-related. A comprehensive discussion on many of these systems can be found in [74]. Though none of them bear similarity to this generic architectural-skeleton model, some of them are worth mentioning here.

HPC++ [26] focuses on a common foundation for portable parallel applications. Parts of its implementation are through libraries and parts through C++ language

extensions. One of its key features is the exploration of loop parallelism, as in HPF [46]. Another feature is the parallel extension of the C++ standard template library (STL) [52]. The parallel standard template library (PSTL) provides distributed versions of the STL container classes along with versions of the STL algorithms that have been modified to run in parallel. As a major distinction with most pattern-based systems, including this architectural-skeleton approach, PSTL is a class-library and not a framework (i.e., the user selects the library routines for an application and it is the user's application that dominates). Other major differences lie in the HPF-like features, such as loop-parallelism and the extended C++ language syntax.

POOMA (Parallel Object-Oriented Methods and Applications) [19] is a collection of C++ template-classes for writing high-performance scientific applications. It provides high-level data-parallel types (for instance, high-level abstractions for multi-dimensional arrays, computational mesh, etc) that make it easy to write parallel PDE (partial differential equation) solvers without worrying about the low-level details of layout, data transfer, and synchronization. In its restricted problem domain, POOMA is able to provide good amount of optimizations for achieving high performance. In comparison, the architectural-skeleton approach is not restricted to any specific problem domain inside parallel computing, and hence, it may not be able to offer the same amount of domain-specific optimizations as in POOMA.

Similarly, DAPPLE [42] is another C++ class library that provides the illusion of a data-parallel programming language on conventional hardware and with conventional compilers. DAPPLE defines Vectors and Matrices as basic classes, with all the C++ operators overloaded to provide for element-wise arithmetic. In addition, DAPPLE provides typical data-parallel operations that are most commonly

applied. In comparison to DAPPLE's exclusive data-parallel domain, the architectural skeleton library is applicable to a much wider range of parallel programming paradigms.

7.4 Future Research Directions

The following are some of the issues that need to be considered in the future evolutions of the work:

- How flexible and usable is the approach? How can these issues be measured and compared? (For instance, consider the software metrics measurements for assessing software complexity). Usability evaluations based on existing usability metrics and statistical experiments need to be planned out in the future.
- Though the PASM model inherently supports extensibility, the issue of extensibility needs to be further investigated for the PASM system. The object-oriented design of the skeleton library needs to be further fine tuned so that adding a new skeleton to the library merely becomes an issue of filling in some pre-defined blanks (i.e., filling in the pre-defined virtual methods).
- Does the PASM system need a graphical user interface (GUI)? Will the introduction of a GUI-based system hamper extensibility? These issues need further investigation.
- The issue of process-processor mapping was mentioned before. Presently no specific mapping strategies are employed, which is a separate research topic

in itself. In future versions, several pattern-specific heuristic-based mapping strategies need to be designed and investigated.

- The generic model can suitably realize all of the frequently used patterns in network-oriented parallel computing that are known at this moment. Are there any other useful patterns left out? Is there any such pattern that cannot be realized inside the model? In that hypothetical latter case, what modifications to the model will be necessary?

  One such pattern is the data-flow pattern which, though useful, is rarely applied in practice. If at all needed, it could be implemented using the compositional skeleton or could be designed as a separate pattern conforming to the model. Another pattern is the client-server pattern, which is in fact a pattern for distributed computing (as compared to the focus of this work, which is on distributed parallel computing).

7.5 Conclusion

The research presents a generic model for designing and developing parallel applications, and is based on the idea of design patterns. The model is based on the message-passing paradigm that makes it well suited for a cluster of workstations and PCs. An architectural skeleton is a physical abstraction of a pattern in parallel computing. The skeleton-based model is an ideal candidate for implementation using object-oriented techniques. The object-oriented approach can be used to build an application-independent, extensible library of skeletons. Other issues of equal importance that form integral parts of the model are: flexibility, reusability (of code for patterns and of application code), separation of specifications, and

inherent support for hierarchical development and refinement.

The present collection of architectural skeletons supports those patterns for coarse-grain message-passing computation which can provide good performance in a networked MIMD environment. Research is in progress to incorporate new skeletons for such an environment.

Bibliography

[1] LAM/MPI Parallel Computing. http://www.lam-mpi.org/.

[2] Message Passing Interface Forum. http://www.mpi-forum.org/.

[3] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.

[4] J. Backus. Can Programming Be Liberated from the Von Neumann Style? A Functional Programming Style and its Algebra of Programs. *Communications of the ACM*, 21(8), August 1978.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformation for High-Performance Computing. Technical Report UCB/CSD-93-781, University of California, Berkeley, 1993.

[6] A. Bartoli, P. Corsini, G. Dini, and C. A. Prete. Graphical Design of Distributed Applications Through Reusable Components. *IEEE Parallel and Distributed Technology*, 3(1):37–50, Spring 1995.

[7] A. D. Birrell and B. J Nelson. Implementing Remote Procedure Calls. *ACM Transaction on Computer Systems*, 2:39–59, Feb. 1984.

[8] G. Booch. J. Rumbaugh. and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Publishing Company. 1999.

[9] P. Brinch Hansen. *Studies in Computational Science: Parallel Programming Paradigms.* Prentice Hall. 1995.

[10] P. Brinch Hansen. *The Search for Simplicity: Essays in Parallel Programming.* IEEE Computer Society Press. Los Alamitos. California. 1996.

[11] J. C. Browne. M. Azam. and S. Sobek. CODE: A Unified Approach to Parallel Programming. *IEEE Software.* pages 10–18. July 1989.

[12] J. C. Browne. S. I. Hyder. J. Dongarra. K. Moore. and P. Newton. Visual Programming and Debugging for Parallel Computing. *IEEE Parallel and Distributed Technology.* 3(1):75–83. Spring 1995.

[13] F. W. Burton and V. J. Rayward-Smith. Worst case scheduling for parallel functional programs. *J. Functional Programming.* 4(1):65–75. January 1994.

[14] D. K. G. Campbell. Towards the Classification of Algorithmic Skeletons. Technical Report YCS 276. Department of Computer Science. University of York. 1996.

[15] N. Carriero and D. Gelernter. How to Write Parallel Program: A Guide to the Perplexed. *ACM Computing Surveys.* 21(3):323–357. September 1989.

[16] K. Mani Chandy. Concurrent Program Archetypes. In *International Parallel Processing Symposium.* 1994. Keynote Address.

[17] K. Mani Chandy. The Caltech Archetypes/eText Project. September 1996. http://www.etext.caltech.edu.

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

[19] J. C. Cummings, J. A. Crotinger, S. W. Haney, W. F. Humphrey, S. R. Karmesin, J. V. W. Reynders, S. A. Smith, and T. J. Williams. Rapid Application Development and Enhanced Code Interoperability using the POOMA Framework. In *SIAM Workshop on Object-Oriented Methods and Code Interoperability in Scientific and Engineering Computing: OO98.*

[20] M. Danelutto and S. Pelagatti. Parallel Implementation of FP using a Template-based Approach. In *Proc. of 5th International Workshop on Implementation of Functional Languages*, The Netherlands, September 1993.

[21] J. Darlington, A. J. Field, and P. G. Harrison. Parallel Programming Using Skeleton Functions. In *PARLE'93*, Munich, Germany, June 1993. Appeared in Lecture Notes in Computer Science, Vol. 694, pages 146-160.

[22] I. Foster and R. Stevens. Parallel Programming with Skeletons. In *ICPP'90.*

[23] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Eaglewood Cliffs, N.J., 1989.

[24] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces^{TM} Principles, Patterns, and Practice.* Addison-Wesley, 1999. The *Jini^{TM}* Technology Series.

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, 1994.

[26] D. Gannon, P. Beckman, E. Johnson, T. Green, and M. Levine. *HPC++ and the HPC++Lib Toolkit.* Department of Computer Science, Indiana Uni-

versity, and Los Alamos National Laboratory. White paper available at http://www.extreme.indiana.edu/hpc++/.

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

[28] A. Geist and V. Sunderam. Network-based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, 4(4):293–311, 1992.

[29] B. Goldberg. Functional Programming Languages. *ACM Computing Surveys*, 28(1):249–251, March 1996.

[30] D. Goswami. Data parallel Solution Strategies for Irregular Problems. Master's thesis, McGill University, June 1995.

[31] D. Goswami, A. Singh, and B. R. Preiss. From Design Patterns to Parallel Architectural Skeletons. *Journal of Parallel and Distributed Computing (JPDC).* Accepted for publication, June 2001. 25 pages. To appear.

[32] D. Goswami, A. Singh, and B. R. Preiss. Architectural Skeletons: The Re-Usable Building-Blocks for Parallel Applications. In *1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '99).* Las Vegas, USA, June 1999.

[33] D. Goswami, A. Singh, and B. R. Preiss. Using Object-Oriented Techniques for Realizing Parallel Architectural Skeletons. In *the third International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE'99)*, San Francisco, USA, December 1999. Appeared in Lecture Notes in Computer Science, Vol. 1732, pages 130-141.

[34] D. Goswami, A. Singh, and B. R. Preiss. Building parallel applications using design patterns. In *Advances in Software Engineering: Topics in Comprehension, Evolution and Evaluation*. Springer-Verlag, New York, 2001. 24 pages. To appear.

[35] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Cambridge, Massachusetts, 1994.

[36] M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, Inc., 1977.

[37] N. Harrison, B. Foote, and H. Rohnert, editors. *Pattern Languages of Program Design 4*. Addison-Wesley Publishing Company, December 1999. Software Patterns Series.

[38] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling FORTRAN D for MIMD Distributed Memory Machines. *Communication of the ACM*, 35(8):66–80, August 1992.

[39] P. Hudak, S. L. Peyton Jones, P. L. Wadler, B. Boutel, J. Fairburn, J. Fasel, M. Gunzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, 27(5), May 1992.

[40] R. E. Johnson and B. Foote. Designing Reuseable Classes. *Journal of Object-Oriented Programming*, June 1988.

[41] O. Kaser, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. EQUALS: a fast parallel implementation of a lazy language. *J. Functional Programming*, 7(2):183–217, March 1996.

[42] D. Kotz. A data-parallel programming library for education(DAPPLE). In *Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, pages 76–81. ACM Press, March 1995.

[43] P. Ledru. JSpace: Implementation of a Linda System in Java. *ACM SIGPLAN Notices*, 33(8):48–50, August 1998.

[44] S. J. Leffer, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of 4.3. BSD Unix Operating System.* Addison-Wesley Publishing Company, Inc., 1990.

[45] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):329–59, 1989.

[46] D. B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, pages 25–42, February 1993.

[47] S. MacDonald, D. Szafron, and J. Schaeffer. Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks. In *5th USENIX conference on Object-Oriented technology and systems (COOTS'99)*, pages 29–43, 1999.

[48] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

[49] G. Meszaros and J. Doble. A pattern language for pattern writing. In *Pattern Languages of Program Design-3*, Software Patterns Series. Addison-Wesley Publishing Company, 1997.

[50] J. J. Modi. *Parallel Algorithms and Matrix Computation.* Clarendon Press, Oxford, 1988.

[51] H. R. Myler and A. R. Weeks. *The Pocket handbook of Image Processing Algorithms in C.* Prentice Hall, 1993.

[52] M. Nelson. *C++ Programmer's Guide to the Standard Template Library.* IDG Books Worldwide, 1995.

[53] R. Pandey and J. C. Browne. A Compositional Approach to Concurrent Programming. In *Parallel and Distributed Programming Techniques and Applications*, pages 1489–1500, 1996.

[54] B. R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in JAVA.* John Wiley & Sons, Inc., 2000.

[55] M. J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill, Inc., 1994.

[56] J. Schaeffer, D. Szafron, G. Lobe, and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, 1(3):85–96, August 1993.

[57] D. C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In 6^{th} *USENIX C++ Technical Conference*, Cambridge, Massascusetts, April 1994.

[58] D. C. Schmidt. Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM (Special issue on Object-Oriented Experiences)*, 38(10), October 1995.

[59] T. M. Shih. *Numerical Heat Transfer.* Hemisphere Publishing Corp., Washington, 1984.

[60] A. Singh, J. Schaeffer, and M. Green. A template-based Tool for Building Applications in a Multicomputer Network Environment. *Parallel Computing 89*, pages 461–466, 1989.

[61] A. Singh, J. Schaeffer, and M. Green. Structuring Distributed Algorithms in a Work-Station Environment: the FrameWorks Approach. In *International Conference on Parallel Processing*, volume II, 1989.

[62] A. Singh, J. Schaeffer, and M. Green. A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):52–67, January 1991.

[63] A. Singh, J. Schaeffer, and D. Szafron. Experience with parallel programming using code templates. *Concurrency: Practice and Experience*, 10(2):91–120, 1998.

[64] S. Siu. Openness and Extendibility in Design-Pattern-Based Parallel Programming Systems. Master's thesis, University of Waterloo, 1996.

[65] S. Siu, M. D. Simone, D. Goswami, and A. Singh. Design Patterns for Parallel Programming. In *Parallel and Distributed Programming Techniques and Applications*, California, August 1996.

[66] S. Siu and A. Singh. Design Patterns for Parallel Computing Using a Network of Processors. In *Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 293–304, Oregon, USA., August 1997.

[67] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *NATO ARW Software for Parallel Computation*, Cosenza, Italy, June 1992.

[68] L. Tahvildari. Assessing the impact of using design-pattern-based systems. Master's thesis, Department of Electrical and Computer Engineering, University of Waterloo, 1998.

[69] L. Tahvildari and A. Singh. Impact of using pattern-based systems on the qualities of parallel applications. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000)*, pages 1713–1720, Las Vegas, USA, June 2000.

[70] S. M. Trewin. PUL-TUF Prototype User Guide. Technical Report EPCC-KTP-PUL-TF-PROT-UG 1.7, University of Edinburgh, July 1993.

[71] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[72] D. Walker. The Design for Standard Message Passing Interface for Distributed Memory Concurrent Computers. *Parallel Computing*, 20(4):657–673, 1994.

[73] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl*. O' Reilly & Associates, Inc., 1996.

[74] G. V. Wilson and P. Lu, editors. *Parallel Programming using C++*. The MIT Press, 1996.