The author hereby grants to VPI & SU permission to reproduce and
to distribute copies of this thesis in whole or in part.

PROBLEM SPECIFIC ENVIRONMENTS FOR
PARALLEL SCIENTIFIC COMPUTING

by

Loretta Sue Auvil

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

©Loretta Sue Auvil and VPI & SU 1992

APPROVED:

Calvin J. Ribbens

Calvin J. Ribbens, Chairman

Clifford A. Shaffer

Layne T. Watson

January, 1992

Blacksburg, Virginia
LD
52.55
18.55
1992
#884
8.2
PROBLEM SPECIFIC ENVIRONMENTS FOR PARALLEL SCIENTIFIC COMPUTING

by

Loretta Sue Auvil

Committee Chairman: Calvin J. Ribbens

Computer Science and Applications

(ABSTRACT)

Parallelism is one of the key components of large scale, high performance computing. Extensive use of parallelism has yielded a tremendous increase in the raw processing speed of high performance systems, but parallel problem solving remains difficult. These difficulties are typically solved by building software tools, such as parallel programming environments. Existing parallel programming environments are general purpose and use a broad paradigm. This thesis illustrates that problem specific environments are more beneficial than general purpose environments. A problem specific environment permits the design of the algorithm, while also facilitating definition of the problem. We developed problem specific environments for a simple and a complex class of problems. The simple class consists of two classic graph problems, namely, all pairs shortest path and connected components. The complex class consists of elliptic partial differential equations solved via domain decomposition. Specific problems were solved with the problem specific environments and the general purpose environment, BUILD, which allows the algorithm to be described with a control flow graph. Comparisons between special purpose environments and general purpose environments show that the special purpose environments allow the user to concentrate on the problem, while general purpose environments force the user to think about mapping the problem to the environment rather than solving the problem in parallel. Therefore, we conclude more effort should be spent on building tools and environments for parallel computing that focus specifically on a particular class of problems.
ACKNOWLEDGEMENTS

First of all, I wish to thank Dr. Michael T. Heath of University of Illinois and Center for Supercomputing Research and Development for encouraging me to pursue an advanced degree in computer science.

Secondly, I wish to thank the United States Air Force for providing financial support through the Palace Knight Program.

Next, and most importantly, I wish to thank my advisor, Dr. Calvin J. Ribbens for his support and guidance throughout this research endeavor. I also wish to thank Dr. Clifford A. Shaffer and Dr. Layne T. Watson for serving as committee members.

Finally, I wish to thank my family—my mom Gladys McDaniel, my step-father Raube McDaniel, my brother Glen Auvil, my sister Julia Auvil and Joyce Kerr, a special lady who is like a mom. I also wish to thank my friends for their encouragement, especially Stacey Ashlund, Elissa Carapella, Arceo Castillo, Valerie Cohen, Jackie Hainer, Peggy Knox, Joe Lavinus, and Tim Ryan.

I wish to dedicate this thesis to the memory of my father Clyde Auvil.
# TABLE OF CONTENTS

1 Introduction.................................................. 1
  1.1 Problem Description...................................... 2
  1.2 Parallel Scientific Programming Paradigms.............. 3
  1.3 Parallel Programming Tools.............................. 5
  1.4 Approach.................................................. 7
  1.5 Thesis Outline.......................................... 8

2 Related Work.................................................. 10
  2.1 BUILD................................................... 11
  2.2 Other Parallel Programming Environments............... 15
  2.3 Other Parallel Programming Tools........................ 21

3 A Simple Problem Class....................................... 24
  3.1 Solution Using BUILD.................................... 27
  3.2 Description of Application Specific Environment...... 33
  3.3 Solution Using Application Specific Environment...... 39
  3.4 Comparisons.............................................. 40

4 A Complex Problem Class...................................... 44
  4.1 Solution Using BUILD.................................... 46
  4.2 Description of the Application Specific Environment.. 53
CONTENTS

4.3 Solution Using Application Specific Environment .................... 61
4.4 Comparisons ...................................................................... 62

5 Conclusions .................................................................... 66
LIST OF FIGURES

2.1 The BUILD parallel programming environment. .................................. 13
3.1 Sequential algorithm for all pairs shortest paths problem. ................. 25
3.2 Parallel algorithm for all pairs shortest paths problem. ..................... 26
3.3 Use of BUILD to solve the all pair shortest path problem. ................. 29
3.4 Code generated by BUILD to solve the all pairs shortest path problem. 31
3.5 Modified code generated by BUILD to solve the all pairs shortest path. 32
3.6 The application specific graph environment—xgraph. ....................... 35
3.7 The drop down menus and popup windows for the xgraph environment. 37
3.8 Use of the xgraph environment to solve an all pairs shortest path problem. 41
3.9 Solution window for the sample problem shown in Figure 3.8. .............. 42

4.1 Parallel algorithm for solving the partial differential equation. .......... 45
4.2 Use of BUILD to solve partial differential equation problem. ............. 48
4.3 Code generated by BUILD to solve the iteration step. ....................... 50
4.4 Modified code generated by BUILD to solve the iteration step. .......... 51
4.5 The application specific partial differential equation environment—xpde. 55
4.6 The drop down menus for the xpde environment. ............................. 56
4.7 The popup windows for the xpde environment. ............................... 57
4.8 The four mapping techniques of the xpde environment. ...................... 59
4.9 Solution window for the Poisson problem described in Section 4.3. ........ 63
Chapter 1

Introduction

The importance of parallel computing has increased tremendously in the last few years. People are attempting to solve complicated problems that would not have been considered years ago, because computers were not capable of solving them in a reasonable amount of time. Since technology has improved, more complex problems can now be solved. While the raw processing speed of individual processors has increased substantially, an extensive use of parallelism is a key element in this improving technology. Obviously, parallel computing is of great interest because important problems can be solved, and solved faster. However, parallel computers are rarely easy to program. Typically, programmers must become experts in programming a specific parallel machine, because they have to learn the system so well to achieve success. Parallel programming has a promising future for scientific computing, so effort must be spent to provide software support for parallel computing. For a discussion of the most pressing software support needs of scientific parallel computing, see [Pancake 91].

The purpose of this chapter is to introduce the general area addressed by this thesis, namely, software environments for parallel problem solving. We describe some of the major problems inherent in parallel programming (Section 1.1), briefly survey common paradigms used by programmers when thinking about parallel programming (Section 1.2), review several broad categories of tools and environments recently developed by other researchers
CHAPTER 1. INTRODUCTION

(Section 1.3), and introduce our specific approach (Section 1.4).

1.1 Problem Description

Parallel programming is very challenging. The development of very different types of parallel machines (e.g., distributed memory and shared memory machines) requires programmers to think about a target machine when developing a parallel algorithm. Advances in hardware development have led to a very diverse set of parallel systems each with its own set of programming primitives. The parallel programmer must use the programming primitives designed for the target system, which virtually eliminates portability. This also complicates the parallel programmer's job, because the target parallel system and the programming primitives must be known before the parallel algorithm can be implemented. Thus, a successful programmer must develop a high level of expertise to use a given parallel system with its unique set of programming primitives.

In addition to the problems caused by the wide variety of machines and primitives, a variety of other problems face the parallel programmer who is trying to capture the power of parallel computers. Some of the additional problems include: determining the inherent parallelism, expressing the parallelism, decomposing the problem into tasks, mapping tasks to processors, scheduling and synchronizing the tasks, load balancing, performance tuning, testing and debugging, and scalability. These problems may be addressed differently depending on the target parallel machines. Determining the inherent parallelism of a problem may be a difficult task, because until recently programmers have had to program sequentially. This means they have also had to think sequentially. Parallel programming requires a change in thinking and it is not always easy to think about programming in a different way. Thus, finding the parallelism for a given problem may be difficult. Determining the parallelism leads to decomposing the problem into tasks that can be mapped to processors. Tasks may not have the same execution time, so it is necessary to distribute tasks across processors in such a way that their work loads are nearly even. Ideally, a distribution with
CHAPTER 1. INTRODUCTION

An equal load for each processor is desired. Task dependencies have to be considered when mapping tasks to processors. This involves developing an efficient execution schedule with processors synchronized according to the task dependencies. The performance of an initial mapping and scheduling of tasks can often be improved by remapping and rescheduling the tasks. Ideally, an efficient system utilizes all processors equally and fully. Since program execution depends on the decomposition, mapping, scheduling, and synchronization of the tasks, a change to any of these can lead to a totally different computation. Therefore, the execution of a parallel program is often nondeterministic, because some systems map and schedule tasks dynamically. This nondeterministic nature of parallel program execution makes testing and debugging difficult. Another problem with parallel programming is scalability, which refers to using more processors efficiently.

Parallel computing has great potential for solving important complex problems; however, for progress to be made in parallel computing it is necessary to find solutions to these parallel programming problems. Researchers are breaking new ground developing software tools for parallel programming. Before turning to a survey of several relevant tools, we first discuss ways in which scientific programmers think about parallel programming. It is useful to present such a discussion because successful parallel programming tools should reflect the paradigms used by their target audience—parallel scientific programmers.

1.2 Parallel Scientific Programming Paradigms

Many people have been successful at parallel programming, even though the development of parallel algorithms is a very complicated task. Parallel programmers have different approaches for designing and developing parallel programs. For instance, according to [Pancake 90], scientific programmers develop parallel programs differently than computer scientists. In order to determine how parallel programmers devise their parallel algorithms, we recently distributed a survey to the readers of NA-NET (a national electronic mail group for numerical analyst). Our purpose was to identify the different paradigms used to formu-
CHAPTER 1. INTRODUCTION

late parallel algorithms. Our survey asked readers to identify the thinking process involved when developing a parallel algorithm; whether they use a graph, some other abstraction, problem specific terms, or a programming language; whether their approach changes for different areas or different architectures; and whether they use any tools. Only twelve lengthy responses were received, so we make no claim of statistical significance. Nevertheless, we believe the responses highlight a few key points.

Parallel programmers think about parallel algorithm development in many different ways. This was evident by the variety of responses received from the survey. Some used a graph based data dependency method, while others used this method only when dealing specifically with a graph problem. One felt that using communicating sequential processes (CSP) [Hoare 78] was efficient for systems that receive inputs and produce outputs after execution; however, some never used abstractions like CSP, or actors [Agha 86]. One person indicated beginning by mentally formulating the algorithm in parallel pseudocode.

The data dependency and control flow methods were popular. These methods were also used in combination with other approaches. Four of the twelve responders use this paradigm at least some of the time.

Approaches that are more problem specific are commonly used, because these approaches reflect the problem most accurately. Five of the twelve responders use a problem specific approach. Some feel that algorithms can be formulated in terms of problem specific data structures which can be decomposed. This method may be viewed as domain decomposition. For instance, a matrix can be decomposed into rows, columns, or submatrices, where each subdivision gets assigned to a processor. This decomposition leads to diagrams, showing the layout of the data structure as mapped to the processors. These diagrams can help determine precedence relationships that occur due to data dependence among the processes. Another type of domain decomposition involves decomposing spatial domains to solve partial differential equations. Each subdomain is assigned to a processor. This decomposition can also use diagrams to illustrate data dependence among the processes.

Other problem specific approaches look at the operations that can be executed on the
CHAPTER 1. INTRODUCTION

data (data structures) in parallel. This leads to trying to reduce the degree of communi-
cation between the processors. This method is similar to another approach that involves
distributed data structures. This approach looks at the operations of the distributed data
structures and tries to reduce the overhead by forming new structures with groupings of
the operations on a distributed data structure.

Another method used is domain flow. Domain flow allows the program to be seen
as a “transformation of data sets.” The programmer is responsible for developing the
algorithm, while a compiler handles the interoperator parallelism, scheduling, mapping,
and synchronization of the parts.

It is clear from even this small survey that many different paradigms exist for developing
a parallel program and that many of these paradigms are problem specific. Software tools are
usually based on a particular programming paradigm. Several survey responses expressed an
interest in software tools for parallel computing, although no one currently indicated using
any software tool for parallel programming. This is not surprising, since many software
tools are still being developed in ongoing research projects. This development of software
tools can help make parallel programming easier, but an important question that needs to
be considered is: Do parallel programming tools reflect the ways that people actually think
about and develop their parallel programs? This is one of the motivating questions behind
the work described in this thesis.

1.3 Parallel Programming Tools

The development of software tools to aid parallel computing is underway. Many researchers
are investigating the development of software tools to help parallel programmers. These
tools focus on solving one or more of the problems mentioned in Section 1.1 for a specific
programming paradigm. The goal of these tools is to make parallel programming easier.
They should help increase understanding, reduce development time, and reduce errors.

The software tools being developed may be grouped into the following broad categories:
CHAPTER 1. INTRODUCTION

parallel programming environments, parallel languages and language extensions, automatic parallelizing compilers, debuggers, and performance monitoring and analyzing tools. Some tools in each of these categories are discussed in greater depth in Chapter 2. These tools are designed to solve one or more of the problems of parallel programming previously mentioned. For instance, parallel programming environments help support parallel programming by integrating many problem solving facilities into a single environment. Some of these facilities include: analyzing dependencies of processes, scheduling tasks, balancing the load, and providing configurations for different machines. The languages provide a convenient way for the programmer to express parallelism. Automatic parallelizing compilers detect the inherent parallelism and create an executable parallel program. Debuggers help parallel programmers find errors in their programs. Performance monitors and analyzers help detect load balancing and performance tuning problems. These tools are general purpose tools that provide a framework for solving the problem. A tool is usually an implementation of one of the programming paradigms, or it implicitly assumes a particular paradigm. So the programmer who uses this tool is limited to solving the problem with this approach.

Although many types of tools have been designed, the main focus of this thesis is on parallel scientific programming environments. According to [Snyder 84], a parallel programming environment is “the collection of all language and operating system facilities needed to support parallel programming integrated together into a single system.” This definition is ambitious and may be more of a grand goal than a definition. The existing parallel programming environments usually fail to meet this definition because they do not include all of the operating system facilities. For example, most environments do not provide a way to compile, link, and execute the program within the environment. These tasks have to be done outside the environment. Most environments provide a way to specify the algorithm in some language. The goal is to make parallel programming easier by incorporating the appropriate facilities into a single environment, so the use of a graphical user interface becomes important. Thus, for our purpose, the definition of a parallel programming environment is modified to be a system that integrates the facilities needed to support parallel programming.
CHAPTER 1. INTRODUCTION

with a graphical user interface. The facilities provided in these environments may vary from one to the next and depend on the parallel problem being tackled and the approach taken to solve this problem. Therefore, no stipulation is made concerning the exact facilities that need to be included.

Most parallel programming environments that have been developed thus far are very general purpose. The general purpose nature of these parallel programming environments provides an approach to the design and specification of parallel programs. The advantage is that they can be used in many application areas, but are they suitable for solving complex problems? Do they help the programmer solve specific applications or are they too general? Questions like these are addressed in this thesis. Perhaps programmers need special purpose tools rather than general purpose tools. An analogy with programming languages can be drawn, where the use of special purpose languages has been more appropriate for certain applications. For example, ELLPACK [Rice 85] is designed for solving linear elliptic partial differential equations, MATLAB [Matlab 89] is designed for matrix computations, and TEX [Knuth 84] is designed for typesetting. Therefore, it seems likely that special purpose environments may be more beneficial for parallel programming.

1.4 Approach

As suggested above, the primary focus of this research is to compare general purpose and special purpose environments for parallel problem solving. Most current parallel programming environments are based on a very general paradigm such as data flow graphs (see the extensive survey in Chapter 2). At the same time, quite problem specific paradigms are often used by successful parallel programmers. Hence, we want to begin addressing questions like the following:

- What would special purpose parallel programming environments look like?

- Will problem specific environments provide enough advantages over general purpose environments to warrant their development?
CHAPTER 1. INTRODUCTION

- How difficult are problem specific environments to build?

- Is there some way that general purpose environments can be more adaptable so that special purpose systems can easily be built on top of them?

The approach used to begin answering these questions is a case study comparison of general purpose and problem specific environments to determine the effectiveness of each. For this case study, two classes of problems were selected—a simple one and a complex one. The simple problem class consists of two classic graph problems; the complex problem class consists of elliptic partial differential equations solved via domain decomposition. The general purpose environment selected to solve each problem is BUILD [Brewer 89]. This environment represents one parallel programming paradigm; other tools exist that use different paradigms. So a discussion of some of the other environments appears in the next chapter. We also designed and implemented a problem specific environment for solving each of the problems. It was necessary to decide what a problem specific environment would look like for each problem. The problem specific environments were developed using the X Window System to create the graphical user interface. The case study compares the different experiences in solving the problems with each environment.

1.5 Thesis Outline

This section describes the organization of the thesis. Chapter 2 contains a section describing the general purpose environment, BUILD, used for making comparisons, and also includes a literature survey of other related software tools. A section is devoted to a discussion of other parallel programming environments and another section to a discussion of other related parallel programming tools. Chapter 3 describes the simple problem class, while Chapter 4 describes the complex problem class. These chapters contain a description of the problems being solved, a discussion of the solution using BUILD, a description of the problem specific tool that was developed, a discussion of the solution using the problem
CHAPTER 1. INTRODUCTION

specific tool, and a comparison of the different environments. Chapter 5 presents results and conclusions of the case study evaluation.
Chapter 2

Related Work

This chapter discusses several well known parallel programming tools that exist today. Parallel programming tools include parallel programming environments, parallel languages and language extensions, automatic parallelizing compilers, debuggers, and performance monitoring and analyzing tools. These tools are developed to help the programmer with programming problems that are inherent to parallel programming. The tools should be easy to use and help improve the understanding of parallel programming. Thus, the tools are very general purpose and not designed to solve any specific problem. Using these tools is usually easier than building a tool from scratch; however, the user may find it necessary to concentrate on mapping the problem to the tool, rather than solving the problem in parallel.

Modern parallel programming environments usually have a graphical interface to make it easier for the programmer to develop a parallel program. These environments may analyze the dependencies among the processes, schedule tasks on the target machine, balance the load among processors, or provide configurations for different machines. Many of these general purpose environments solve some subset of the programming problems faced by parallel programmers. Section 2.1 is devoted to a discussion of the general purpose parallel programming environment, BUILD, which we used for our comparisons. Section 2.2 describes
CHAPTER 2. RELATED WORK

other parallel programming environments. The final section of this chapter discusses other related parallel programming tools.

2.1 BUILD

BUILD [Brewer 89] was developed at Argonne National Laboratory to provide an interface to the FORTRAN language extension, SCHEDULE [Dongarra 89]. The goal of BUILD and SCHEDULE is to minimize the effort required to port an algorithm to a specific parallel machine. The availability of SCHEDULE on several parallel machines eliminates the need to learn the parallel programming primitives for each of these machines. This makes SCHEDULE a general purpose package that assists in the development of portable parallel programs.

SCHEDULE by itself is just a FORTRAN language extension where a library of subroutine calls adds the parallel constructs necessary to implement parallelism. These parallel constructs control the spawning, mapping, and synchronizing of processes given a list of processes on which each process depends. SCHEDULE will schedule a process to begin execution when all the processes on which it depends have completed and a processor is idle. This is the control flow approach of parallel programming, where the programmer formulates an algorithm by identifying the flow of control through the algorithm. When using SCHEDULE to design an algorithm one needs to identify the processes that must complete before a given process may begin. So the development process usually includes drawing a dependency (or control flow) graph, where the nodes represent the processes and the arcs represent the precedence relationships. At this stage of development BUILD becomes useful.

BUILD is a graphical front end that makes SCHEDULE a parallel programming environment. It provides an interface for controlling the development of the algorithm. BUILD helps the programmer identify the processes and specify the precedence relationships by providing facilities for labeling and drawing the dependency graph. FORTRAN code with
CHAPTER 2. RELATED WORK

SCHEDULE calls can be generated from the dependency graph specified from within BUILD. However, this code is not executable; the user must add variable declarations, initializations, and subroutine parameters. BUILD does not provide help with designing the subroutines for the processes; it only provides a facility to pull the separate subroutines together. The programmer is responsible for writing the subroutines.

The BUILD interface with a typical dependency graph is illustrated in Figure 2.1. It is composed of two basic parts: the control panel and the drawing pad. We describe the features of the control panel before describing how a dependency graph is drawn and manipulated. A dependency graph can also be saved in a graph file so that it may be retrieved later. The buttons and switches in the control panel are activated by clicking the left mouse button. The Directory and Graph file buttons help the user to find the desired graph file that is to be retrieved. The Subroutine button displays the names and parameter lists of the FORTRAN subroutines found in the file sub.list. The user can cycle through these subroutine names to locate the desired subroutine. The Current node window indicates the last node that was manipulated. The Node Placement switch can be either automatic, where the program controls the placement of the nodes, or user, where the user controls the placement of the nodes. The Set Subroutine switch is used to assign a process to a node and to display the subroutine name of a node. When this is off and the right mouse button is clicked with the cursor on a node in the drawing pad, the subroutine name of this node is displayed in a box to the right of the node. When the Set Subroutine is on, the subroutine name displayed in the subroutine button is assigned to the node that is chosen by clicking the right mouse button with the cursor on that node. The Erase node/arc switch allows a single node or arc to be deleted. The Node Type switch can be either regular indicating one computation of a subroutine or loop indicating n computations of a subroutine. When using the loop node type, n is a variable that can be initialized or specified at run time by modifying the parallel code that was generated.

The Load file and Save graph buttons are for retrieving and saving a graph, respectively. The Dump code button is for saving the code generated to a file. The Names button
Figure 2.1: The BUILD parallel programming environment.
CHAPTER 2. RELATED WORK

displays the subroutine name of each node. The Edit button lets the user edit a subroutine associated with a particular node. The user clicks the left mouse button with the cursor on the Edit button and then clicks the left mouse button with the cursor on the desired node. A window running the vi editor pops up containing the subroutine code for this node. The subroutine code is retrieved and stored with the subroutine name followed by the .f extension. The user can then make modifications to the subroutine. No other commands can be executed while a subroutine is being modified. Some of the other buttons that are relatively straightforward include: Clear screen, Redraw, and Quit.

A dependency graph can be specified in the drawing pad by using the control panel and the mouse. When the left mouse button is clicked with the cursor in the drawing pad, a node is added at the cursor position. A node can be moved by clicking and holding the left mouse button with the cursor on that node, moving the cursor to the desired location, and releasing the button. A node can be deleted by setting the Erase node/arc switch to on and then clicking the left mouse button with the cursor on the node to be deleted. An arc can be added by clicking and holding the middle mouse button with the cursor on the desired child node, moving the cursor to the desired parent node, and releasing the button. An arc can be deleted by setting the Erase node/arc switch to on and then clicking and holding the middle mouse button with the cursor on the desired child node, moving the cursor to the desired parent node, and releasing the button. As described above, the right mouse button is used for assigning and displaying subroutine names. This was just a brief description of the features of BUILD. For more detail, see [Brewer 89].

SCHEDULE also provides a postprocessing performance analysis of the parallel algorithm. This analysis is an animation of the flow of the execution of the processes. Trace information is collected and stored in a file while the parallel program is executing. The performance analysis tool uses the trace information to animate the flow of the execution of the processes. This can reveal performance bottlenecks and possible logic errors, but we did not experiment with the trace facility in this research.
CHAPTER 2. RELATED WORK

2.2 Other Parallel Programming Environments

This section discusses some parallel programming environments, other than BUILD, and describes the way some of these other environments help the parallel programmer. Some of these use the same programming paradigm as BUILD while others use a different one, but most use either the control flow or data flow paradigm. Each of these environments provides help with a different set of parallel programming problems.

CODE and ROPE

CODE (Computation-Oriented Display Environment) [Browne 89, Jain 91], and ROPE (Reusability-Oriented Programming Environment) [Browne 90] were developed at the University of Texas. The programming paradigms used in this project are the control and data flow approaches. The goal of this project is to develop a programming environment where specification of a program is independent of its architecture and execution environment. This project merges parallel programming with visual programming and software engineering. CODE provides a visual and form-oriented programming interface based on Unified Computation Graphs (UCGs). UCGs provide a means for describing parallel computations at a high level of abstraction, because they separate the specification of algorithms from their target architecture. A UCG has nodes and edges representing computations and dependency relations, respectively. CODE provides forms for describing each of the nodes and arcs of the graph. ROPE fosters good software engineering practice by focusing on the reuse of software components. The declarative, hierarchical, and graphical programming nature of CODE allows ROPE to implement capabilities for finding, understanding, modifying, and combining these components. The result is that ROPE usually decreases program development time and error rates. A parallel program can be written by first having the programmer draw the computation graph and define the dependency relationships. The name of the file containing the procedures needs to be specified, as well as the programming language. Next, CODE generates the procedure name, names of the dependencies,
CHAPTER 2. RELATED WORK

and a statement returning control. The architecture independent program specifications are created. Finally, a translator maps these specifications to an execution environment. This includes creating tasks, and managing communication and synchronization.

Faust

Faust [Guarna 89] was developed at the Center for Supercomputing Research and Development at the University of Illinois to provide an interactive compiler, optimization tools, and debugging and analysis facilities. This tool was designed to provide a multi-user, distributed and heterogeneous environment. Therefore, a Project Manager was implemented with a locking facility to allow project objects to be shared. The Project Manager is responsible for organizing the objects of a project by creating a directed graph with nodes representing the objects and the arcs representing the relationship between the objects. The Project Manager is also responsible for maintaining all the files used by Faust. Another tool that is implemented in Faust is Sigma, a text editor that interacts with the program database. Sigma is the tool that retargets and optimizes code. Sigma can analyze code for data dependencies, so it can be used to help detect the parallelism within the code. Faust also contains a dynamic call-graph tool, which animates the execution of the parallel program. Impact is Faust’s tool for collecting, analyzing, and displaying performance results. This is another environment that uses the control flow approach.

FrameWorks

FrameWorks [Singh 90] was developed at the University of Alberta, Canada to design distributed programs on a network of workstations. The programming paradigm used for this environment is control flow. This tool uses the remote procedure call (RPC) model to meet the needs of developing a distributed application. FrameWorks allows specification of modules to be separate from the interface, interconnection, and processor allocation. This tool generates the low-level code for synchronization, communication, and scheduling. This tool uses templates and an interactive graphical user interface to specify synchroniza-
CHAPTER 2. RELATED WORK

tion and communication activities. The templates can be combined to construct pipelines, fan-in structures, fan-out structures, and master and slave relationships that specify the communication between modules. A module is defined by its input template, output template and an optional body template. The input and output templates describe the input and output communications for each module. The optional body template may be used to adapt the module's runtime behavior to the distributed environment. The user codes the modules in an extended version of C, which implements statements for passing information to and receiving information from other procedures. This tool also assigns the processes to processors. When the program is executed with the Execution_Time_Monitor facility running, the program terminates when this facility detects deadlock or the termination of a distributed process. The templates can be combined to create different configurations, which can execute the same source code. This provides an effective way to compare the different configurations.

Phred

Phred [Beguelin 96] was developed at the University of Colorado. Phred is a visual parallel programming language that illustrates the parallelism, data flow, and control flow of a program. It provides a graphical interface, static analysis tools, and an interpretive execution environment. The graphical interface allows the programmer to specify Phred program graphs. Phred program graphs consist of nodes representing a task, repositories for storing data, and edges representing precedence and data flow. Edges between task nodes represent precedence, and edges between task nodes and repositories represent data flow. This environment uses both the control and data flow programming paradigms. The static analysis tool, the critic, determines if a graph is a correct Phred graph and analyzes the graph for determinacy. The interpretive execution environment maps the tasks to a parallel machine, passes tokens between tasks, calls user functions when a node can execute, and manipulates the data repositories.
CHAPTER 2. RELATED WORK

PIE

PIE (Programming and Instrumentation Environment) [Lehr 89, Segall 85] was developed at Carnegie-Mellon University. The programming paradigm used in this environment is a formal specification technique for developing parallel programs. The goal of this project was to support the development of performance-efficient parallel programs. It supports the development of code as well as analysis, verification and validation of performance. PIE also proves to be a good performance debugging environment by helping to detect and isolate performance problems. The three components that make up PIE are the modular programming metalanguage (MP), the program constructor (PCT), and the implementation assistant. The MP supports the design of efficient parallel modules and fast parallel access to shared data constructs. It allows parts (or modules) of the program to be specified and coded without knowing the entire computation. The PCT provides a higher level of abstraction than the MP, because the PCT helps with the detailed specifications. This component contains three elements: the MP-oriented editor, status and performance monitor, and a relational representation system. The MP-oriented editor is a syntax directed editor for specifying and coding the problem solution. The status and performance monitor uses sensors, special objects inserted into the code to monitor the program execution. The relational representation system provides different representations to allow the programmer to understand the interactions and bottlenecks of the program. The implementation assistant provides support for semantics in the program development cycle. It helps predict the performance of a parallel program, and it also helps select a way to decompose the parallel computation.

POKER

POKER [Snyder 84] was developed at the University of Washington as an environment for designing and executing parallel programs. The programming paradigm used by this environment is the data flow method. This tool addresses five characteristics common to most parallel algorithms for distributed systems. The first characteristic is the definition
CHAPTER 2. RELATED WORK

of the communication structure of the algorithm. One of the best ways to describe this communication is by a graph where nodes represent processors and edges represent the communication links. POKER enables the user to describe the communication structure of the algorithm by embedding a graph into a lattice. The second characteristic is the description of the computations of each process. POKER provides a primitive sequential programming language, called XX, with data types, arithmetic and logical operators, and control structures. The third characteristic is the naming of the processes. This tool allows processes to be assigned to specific nodes of the graph. The fourth characteristic is the synchronization of the processes. Synchronization of the processes occurs automatically by using a code optimization facility for analysis and conversion of the program. Communication between processes requires that a port name be given where every edge meets a node. The fifth characteristic is the input and output facilities that specify the data flow to and from each node. POKER provides facilities for specifying these five characteristics of parallel programming to develop a parallel program that can be compiled, assembled and linked. During the execution of a program, variables may be traced and changed during suspension with the new values being used after interruption. POKER allows complex problems to be divided into subproblems that can be defined by one communication graph called a phase. These phases can be combined to create more complicated programs.

PPSE

PPSE (Parallel Programming Support Environment) [Lewis 89, Rudd 89] was developed at the Oregon Advanced Computing Institute. The programming paradigm used in this environment is a data flow approach. This environment is designed to parallelize a sequential program and to aid the design, implementation, testing and performance evaluation of a parallel program. PPSE includes the following tools: a graphical design editor, a target machine editor, a mapping heuristic, a scheduler, and a program restructurer. The graphical design editor allows the programmer to specify the parallel algorithm. It is based on ELGDF (Extended Large Grain Data Flow), where a graph is a directed network of
CHAPTER 2. RELATED WORK

nodes (representing processors), storage constructs, arcs (representing data or control dependency), and constructs for loops, pipes, and conditional branching. The target machine editor allows the programmer to specify the target machine with a graphical description language. The mapping heuristic maps the processes to the hardware and produces a so-called Gantt chart, which consists of a list of all processes allocated to each processor and ordered by execution time. The scheduler optimizes performance and processor utilization. The program restructurer parallelizes sequential FORTRAN programs. Another tool is SuperGlue, which combines code fragments, flow file, Gantt chart, and target machine file. The programmer must write the code fragments for the program in terms of procedures. The flow file contains information about procedural connectivity and shared variables. The target machine file contains information about the target architecture, processors available, and programming language. SuperGlue generates the complete parallel program in Linda. Since the architecture is specified separately the programmer can easily change between architectures. The new parallel program can be created by specifying the new target architecture and rerunning SuperGlue. PPSE also includes performance monitoring and debugging tools.

R^n and ParaScope

R^n [Callahan 88] was developed at Rice University to assist in the development of numerical software systems. R^n is an automatic compiler. It consists of five components: a module editor, a composition editor, a module compiler, a program compiler, and an execution monitor. The module editor is used to edit source programs. The composition editor defines source modules in a program. The module compiler compiles and optimizes a program based on interprocedural information. The program compiler computes interprocedural information, performs optimizations, and rebuilds the program after a change. The execution monitor runs programs and provides a debugging facility. This environment does not provide support for explicit parallel programming, which the developers feel is necessary to achieve high performance on parallel machines.

An enhancement to the R^n programming environment is ParaScope [Callahan 88]. ParaS-
CHAPTER 2. RELATED WORK

cope is designed to help formulate, implement, and debug parallel programs. The three areas of enhancement include a source editor, a compiling system, and a debugging system. The source editor is a newer version of the editor for the R^n environment. It allows editing of source programs, examining of the dependence structure of the code, applying prepackaged transformations to code, and viewing the effect of these changes on the dependence structure. The compiling system provides more powerful forms of interprocedural analysis, whole program optimizations, and an expansion of the module compiler to generate parallel code for a target machine. The debugging system is improved by adding remote debugging and parallel debugging. ParaScope also implements a performance visualization tool.

2.3 Other Parallel Programming Tools

This section describes some of the other parallel programming tools that have been developed, such as parallel languages and language extensions, automatic compilers, debuggers, and performance monitoring and analysis tools. These tools are designed to solve some of the problems of parallel computing.

Numerous parallel languages and language extensions have been especially developed for parallel programming. These languages incorporate constructs and features that make explicit parallel programming possible. These constructs and features allow processes to be created and controlled by synchronization and communication mechanisms. OCCAM [Pountain 86], based on Hoare’s communicating sequential processes (CSP), is a good parallel language to use when tasks receive input, update their state, and produce output. Ada [Goos 83] is another parallel language based on Hoare’s communicating sequential processes. In Ada processes synchronize and communicate by using input and output statements. This synchronization and communication process is called a rendezvous. Linda [Carriero 89] is a parallel language extension based on tuple space, logically shared memory. Linda provides a means to create processes by creating a process tuple that becomes a data tuple in tuple space after it is evaluated. Processes communicate by adding and removing tuples from
CHAPTER 2. RELATED WORK

tuple space. This communication process handles synchronization of processes by blocking when a tuple is not present. Nonblocking functions are also available.

Automatic parallelizing compilers or optimizers are implicit parallel programming tools. The task of determining parallelism is removed from the programmer and placed in the hands of the designer of the compiler. The compiler writer must create a tool that can generate parallel code from sequential code. These tools usually provide data dependency analysis, interprocedural analysis, and control and data flow analysis. Partitioning tasks and automatic scheduling are a few other tasks. PARAFASE [Polychro 90] is an example of an automatic vectorizing compiler. It parallelizes sequential C or FORTRAN code. PTRAN (Parallel TRANslator) [Allen 88] and PFC (Parallel FORTRAN Converter) [Callahan 88] automatically restructure and vectorize sequential FORTRAN programs for execution on a parallel machine.

Parallel debuggers are needed to help parallel programmers find errors in their programs, because bugs can be much harder to detect in parallel programs than in sequential programs. Bugnet [Wittie 89] and the Agora debugger [Forin 89] are parallel debuggers that improve error detection by rolling back the execution and replaying the events. PPD (Parallel Program Debugger) [Miller 89] helps the programmer locate the time at which a bug occurred by tracing back through events from the time at which the error was detected. This traceback, called flowback analysis, provides information on the causal relationships of events during execution, so the program is not re-executed during debugging.

Other parallel tools include performance monitors and analyzers, designed to help the programmer study algorithm performance. Tools of this type are usually based on a graphical animation of the program's execution, which could be a visualization of memory accesses, critical path analysis, or communication and event monitoring. These tools may also help detect or predict race conditions and deadlock, and thus, can be quite useful for analyzing the performance of an algorithm. Some examples of performance monitors include Paragraph and IPS-2. Paragraph [Heath 91] is a graphical display system that provides a visualization of the behavior and performance of parallel programs. Many different displays
CHAPTER 2. RELATED WORK

are available to permit the programmer to view a parallel program from many different perspectives. The IPS-2 [Miller 90] provides performance monitoring techniques to help the user find performance problems, like bottlenecks. Some examples of performance analyzers include PARET and PAWS. PARET (Parallel Architecture Research and Evaluation Tool) [Nichols 88] studies the interaction of algorithms and architectures. It allows a programmer to see the results of changing the physical resources on system performance, as well as the effects of changing the mapping, scheduling, and routing strategies. PAWS (Parallel Assessment Window System) [Pease 91] helps the programmer compare and investigate various architectures for a given problem, so the programmer can identify the best architecture for a given problem.
Chapter 3

A Simple Problem Class

The purpose of this chapter is to describe the simple problem class. Section 3.1 describes how BUILD was used to solve the problems. Section 3.2 describes the application specific environment xgraph that we built. Section 3.3 describes the use of the xgraph environment. Section 3.4 contains a comparison of the two environments.

The simple problem class consists of two classic graph problems, namely, the all pairs shortest path problem and the connected components problem. These two problems are very similar. The all pairs shortest path problem determines the length of the shortest path between every two points in a graph. The connected components problem determines whether a path exists between every two points in a graph. Both problems can be solved for a directed or an undirected graph. The undirected graph can be represented as a directed graph with two directed edges in opposite directions for each undirected edge.

The algorithms that we present were developed for a shared memory machine. The shared memory approach makes the data available to all processors, so the data does not have to be explicitly assigned to processors. However, for a distributed memory machine, the data that a particular processor needs in order to solve its portion of the problem needs to be assigned to that processor or communicated to that processor via a message. Therefore, the algorithms for solving these graph problems on a distributed memory machine could be
CHAPTER 3. A SIMPLE PROBLEM CLASS

<table>
<thead>
<tr>
<th>Input: weight (an n x n adjacency matrix representing a weighted graph). weight[i, j] is the weight of the edge (i, j) if it exists, or ∞ otherwise; weight[i, i] is 0, for all i.</th>
<th>Output: the matrix weight containing the lengths of the shortest paths.</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin for k := 1 to n do</td>
<td>begin for i := 1 to n do</td>
</tr>
<tr>
<td>for j := 1 to n do</td>
<td>if weight[i, k] + weight[k, j] &lt; weight[i, j] then</td>
</tr>
<tr>
<td>weight[i, j] := weight[i, k] + weight[k, j]</td>
<td>end</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.1: Sequential algorithm for all pairs shortest paths problem.

different than the shared memory algorithms.

The all pairs shortest path problem consists of finding the shortest path lengths between all pairs of points in a graph with nonnegative weights associated with each edge. This weighted graph can be represented as an n x n weighted adjacency matrix (referred to as weight), where n is the number of points in the graph. Initially, weight[x, x] is 0, and weight[x, y] is the weight of the edge from point x to point y if it exists, otherwise, infinity. After executing the all pairs shortest path algorithm, the adjacency matrix weight contains the lengths of the shortest paths between all points. The algorithm checks all pairs of points to see if a shorter path exists by going through another point rather than using the shortest currently known path between the pair of points. This means determining whether the path length from point i to point k plus the path length from point k to point j is shorter than the current path length from point i to point j. The sequential algorithm is given in Figure 3.1. This algorithm uses three loops. The outer loop checks whether a path going through point k shortens the path between any two points. The inner two loops are used to check all pairs of points.

The algorithm given in Figure 3.1 is for solving a directed graph problem. The solution to a directed graph problem requires that every pair of points be checked in both directions—
CHAPTER 3. A SIMPLE PROBLEM CLASS

Input: weight (an $n \times n$ adjacency matrix representing a weighted graph).
weight[i, j] is the weight of the edge (i, j) if it exists,
or $\infty$ otherwise;
weight[i, i] is 0, for all i.
Output: the matrix weight containing the lengths of the shortest paths.
begin
  for $k := 1$ to $n$ do
    for $i := 1$ to $n$ do in parallel
      for $j := 1$ to $n$ do
        if weight[i, k] + weight[k, j] $<$ weight[i, j] then
          weight[i, j] := weight[i, k] + weight[k, j]
  end

Figure 3.2: Parallel algorithm for all pairs shortest paths problem.

from point $i$ to point $j$ and from point $j$ to point $i$. When an undirected graph problem is being solved each pair of points only has to be checked once. Since direction of an edge does not matter, the solution is the same in both directions. The only change in the algorithm is that the innermost loop does not need to check from 1 to $n$, but rather only from $i$ to $n$.

This problem was selected because it lends itself to a straightforward parallel algorithm. The check for all pairs of points can be applied in any order, because each check is independent of the others. Therefore, the checks for all paths from different initial points going through point $k$ can be done in parallel. This means the second loop can be done in parallel. The parallel algorithm is given in Figure 3.2. Again this algorithm is for a shared memory machine and takes advantage of the fact that all processors can access the data. When using a distributed memory machine, the algorithm would have to reflect the fact that the data needs to be distributed to the processors.

We have described the all pairs shortest path problem in great detail. Since the connected components problem is similar, a description of the differences between the two algorithms will provide sufficient explanation. The connected components problem determines the existence of a path between every pair of points in the graph. We also use an adjacency matrix weight to solve this problem. Initially, weight[x, x] is one, and weight[x, y] is one if the edge from point $x$ to point $y$ exists, zero otherwise. After executing the con-
CHAPTER 3. A SIMPLE PROBLEM CLASS

connected components algorithm, the adjacency matrix indicates the existence of a path with a one. The algorithm checks all pairs of points to see if a path exists between two points by going through another point. This means determining whether the path from point \( i \) to point \( j \) exists by checking to see that the path from point \( i \) to point \( k \) exists and the path from point \( k \) to point \( j \) exists. This algorithm can be optimized by only executing the innermost loop when the path from point \( i \) to point \( k \) exists. The discussion of directed and undirected graphs applies to this algorithm in the same way, that is, the undirected graph need only be solved in one direction. This algorithm can be parallelized in the same manner as the all pairs shortest path problem.

3.1 Solution Using BUILD

This section describes how BUILD can be used to solve our simple graph problems in parallel. We have chosen to describe a solution to the all pairs shortest path problem for a directed graph. Solutions to an undirected graph problem and the connected components problem are very similar. Therefore, a detailed description of a solution for one of the problems will suffice.

The BUILD environment allows one to think in terms of control flow. So the parallel algorithm can be specified in BUILD by specifying a control flow graph. This means that the programmer must determine where this problem can be parallelized. In this situation, we must test the path through different points sequentially. The pair checking of points \((i, j)\) for a given point \( k \) may occur in parallel. When we begin to draw the control flow graph, we may use either regular nodes (specifying one computation) or loop nodes (specifying \( n \) computations). It is possible to solve this particular problem with either method, although using regular nodes may result in a more complicated dependency graph. Since we are using loops in our algorithm, it is appropriate to use the loop capability that BUILD has provided. Although BUILD does provide a loop capability, it does not allow one to draw a control flow graph for a loop within a loop. Thus, we have to draw a loop node for every
CHAPTER 3. A SIMPLE PROBLEM CLASS

point in the graph, so each loop node corresponds to an iteration of the first loop of the algorithm shown in Figure 3.2. Each loop node will spawn \( n \) identical tasks which can execute in parallel. Each of these spawned tasks corresponds to an iteration of the second loop in the algorithm shown in Figure 3.2. The control flow graph defined using BUILD is illustrated in Figure 3.3.

The programmer is responsible for designing the algorithm, which includes determining a representation for a specific problem. In this case, we must decide how to represent the initial problem graph. We decided that the initial problem graph would be defined by the number of points in the graph and a list of edges (represented by a pair of points) with weights. This information can be translated into a matrix representation during initialization.

A subroutine needs to be specified for each node in the control flow graph. BUILD allows the programmer to edit a file for a node by clicking with the cursor on the Edit button and then clicking with the cursor on the node. This brings up a window running the \( vi \) editor. The programmer can then specify or modify the code for this subroutine. For this problem, each node will execute the same subroutine, so it is only necessary to write the subroutine once and then assign the subroutine to the other nodes in the control flow graph. Subroutine names can be assigned by creating or appending the subroutine name and parameter list to the sub.list file, which must be done outside the BUILD environment. Hence, a programmer needs to know what the subroutine names and parameter lists will be before entering the BUILD environment to generate the code. The programmer may enter the BUILD environment, draw the dependency graph, save the dependency graph, exit the environment, add the subroutine names and parameters to the sub.list file, and then reenter the environment. Upon reentering the environment the programmer may assign the subroutine names to the nodes of the control flow graph. Since the sub.list file now contains the subroutine names and parameter lists for the problem to be solved, the programmer can assign subroutines to nodes. Subroutines are assigned to nodes by setting the Set Subroutine switch to on, selecting the correct subroutine by cycling through the subroutine names displayed in the Subroutine window by clicking the left mouse button,
Figure 3.3: Use of BUILD to solve the all pair shortest path problem.
CHAPTER 3. A SIMPLE PROBLEM CLASS

and finally clicking the right mouse button with the cursor on the node in the control flow graph. After assigning a subroutine name to each node, the code can be generated by clicking with the cursor on the Dump code button and specifying a file name. Before exiting from the environment, the user may save the control flow graph by clicking the left mouse button with the cursor on Save graph and specifying a file name.

The BUILD environment has now fulfilled its goal, generating code with the SCHEDULE calls which handle the spawning, mapping, and synchronizing of tasks. However, the code generated by BUILD, shown in Figure 3.4, is not executable.

The programmer must add variable declarations, initializations, subroutine parameters, and possibly output statements. Some of the variables that SCHEDULE uses need to be placed in COMMON blocks, and the variables needed to solve this problem also need to be declared. Some declarations need to be made in practically every subroutine. The initialization for this problem includes describing a representation for the problem graph. The problem graph is specified by identifying the number of points in the graph, the edges, and the weight of each edge. An edge is defined by specifying two points—the source and destination. BUILD does not provide a way to select the number of processors to be used to solve this problem, so the number of processors nproc must be initialized or programmed to be defined at run-time. Since the loop node is used, it is also necessary to initialize the number of iterations numits of the loop. The number of iterations of the loop may also be defined at run-time, but code has to be added to make this possible. Subroutine parameters have to be added for the subroutines the programmer defined. The subroutine parameters have to be changed for the loop subroutine that BUILD created for the loop nodes, because SCHEDULE overwrites the subroutine parameters when the next loop call is made. Therefore, an indexing method to retain the parameter values is necessary. The only other change is adding output statements, so that the results can be stored in a file. The modified code is shown in Figure 3.5 with changes in italics. The program can now be compiled, linked, and executed.

Using the BUILD environment to solve the connected components problem is very sim-
external paralg
call sched(nproc,paralg)
stop
end
subroutine paralg
external loop1
integer jobtag, icango, nchks, mychks(1)
integer numits
   jobtag = 1
   icango = 1
   nchks = 0
   call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, loop1, jobtag, numits)
   .
   .
   jobtag = 16
   icango = 0
   nchks = 1
   mychks(1) = 15
   call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, loop1, jobtag, numits)
return
end
subroutine loop1(myid,n)
integer myid, n
logical wait
external algorithm
go to (1100,1200), ientry(myid,2)
1100 continue
do 1150 i = 2, n
   call nxtag(myid, jdummy)
call spawn(myid, jdummy, algorithm)
1150 continue
call algorithm
   if (wait(myid,2)) return
1200 continue
return
end

Figure 3.4: Code generated by BUILD to solve the all pairs shortest path problem.
external paralg, init, save_output
COMMON /Glob/ numits, index(20), num_points, paths(20, 20)
integer paths
call init() including nproc = 8 and numits = num_points
call sched(nproc,paralg)
call save_output()
stop
end

subroutine paralg
COMMON /Glob/ numits, index(20), num_points, paths(20, 20)
external loop1
integer jobtag, icango, nchks, mychks(i), numits, paths
    jobtag = 1
    icango = 1
    nchks = 0
    call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, loop1, index(jobtag), numits)
    : 
    jobtag = 16
    icango = 0
    nchks = 1
    mychks(i) = 15
    call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, loop1, index(jobtag), numits)
return
end

subroutine loop1(myid,n)
COMMON /Glob/ numits, index(20), num_points, paths(20, 20)
integer myid, n, paths
logical wait
external algorithm
go to (1100,1200), ientry(myid,2)

1100 continue
do 1150 i = 2, n
    call nxtag(myid, jdummy)
    call spawn(myid, jdummy, algorithm, index(myid), index(i))
1150 continue
    call algorithm(index(myid), index(1))
if (wait(myid,2)) return
1200 continue
return
end

Figure 3.5: Modified code generated by BUILD to solve the all pairs shortest paths.

32
CHAPTER 3. A SIMPLE PROBLEM CLASS

ilar. The dependency graph is the same, but the subroutines assigned to each node are slightly different, reflecting the existence of a path rather than the length of the path. Other than this difference, solving the connected components problem is done the same way as the all pairs shortest path problem.

BUILD handles the spawning, mapping, and synchronizing of the tasks; however, having to make changes to the generated code is a handicap to programmers using this environment. The programmer must understand the code which is generated by BUILD, and this entails understanding the SCHEDULE calls. Next we consider what an application specific environment to solve these classic graph problems looks like.

3.2 Description of Application Specific Environment

This section describes the application specific environment, xgraph, that is used to solve (in parallel) the graph problems addressed in this chapter. The xgraph environment was built using the Athena widget set provided in the X Window System. An application specific environment provides the appropriate facilities to solve a problem in parallel by specifying the algorithm and defining the problem. We begin this section by identifying the important characteristics of this problem that an environment needs to address.

Since this environment is for a specific application, specifying the algorithm may only require selecting from a menu items which determine characteristics of the algorithm. The generic form of the parallel algorithm is built into the system. In this situation, the specific instance of the algorithm may be specified by identifying the type of problem, the type of graph, and the number of processors. The type of problem may be either the all pairs shortest path problem or the connected components problem. The type of graph may be either a directed or an undirected graph. The environment can be developed to easily apply to both problems and both graph types. The algorithm reflects the problem and graph type. When the graph is undirected, the problem does not have to be solved from point \( a \) to point \( b \) and point \( b \) to point \( a \). The undirected nature of the graph means that it is only
CHAPTER 3. A SIMPLE PROBLEM CLASS

necessary to solve the problem in one direction, whereas the directed graph problem has to be solved in both directions. The number of processors can also be specified. Setting the type of problem, the type of graph, and the number of processors specifies the algorithm. Next, we look at how a problem can be defined.

These are graph problems, and therefore are naturally defined in terms of a graph. It is plausible that the environment provide the capabilities necessary to draw a graph, and therefore the xgraph environment includes facilities to draw and erase points and edges. When solving the all pairs shortest path problem, after an edge is drawn, the user is asked to enter a weight for that edge. The xgraph environment also provides a facility to change the weight of an edge for the all pairs shortest path problem. It is not necessary to specify a weight or change a weight for an edge in the connected components problem, because the weight of every edge is one. When a directed graph is selected, the lines between points have an arrow pointing toward the destination point. When an undirected graph is selected, lines are drawn between points.

The application specific environment xgraph is shown in Figure 3.6. The Problem button is a drop down menu that allows the user to select either the all pairs shortest path problem or the connected components problem. The Graph Type button is a drop down menu that allows the user to select either a directed or an undirected graph. The Number of Processors button is a popup window that enables the user to set the number of processors by adjusting the slider to the desired setting with the middle mouse button. The Output button is also a drop down menu that allows the user to store the solutions in a file or to display them in a window. Each of these menus are shown in Figure 3.7. Default values are set for each of these selections, so the user may enter the application and begin drawing the graph if the default values are appropriate for this particular problem. The default values for these four menu selections are the all pairs shortest path problem, an undirected graph, one processor, and a window.

The Solve button executes the program, reflecting the selections made defining the algorithm and the graph that is currently displayed in the drawing area. The solutions,
Figure 3.6: The application specific graph environment—xgraph.
CHAPTER 3. A SIMPLE PROBLEM CLASS

presented as an adjacency matrix for both graph problems, are either displayed in a window or stored in the file specified. The adjacency matrix for the all pairs shortest path problem shows the length of the path (with infinity shown as a dash). The connected components problem shows the adjacency matrix with one for an existing path, otherwise, zero. The Help button was added to provide help to a novice user. The help information is displayed in a popup window providing scrolling. The Quit button exits the application environment. The long narrow box is used to display messages during execution of the environment. Useful messages are displayed to indicate what the system is doing.

The Load Graph and Save Graph buttons are useful facilities for reusability. The Load Graph button allows a graph file that has been saved to be loaded into the system. This command provides a popup window asking the user to enter the name of the graph file. When an existing graph file is entered, the current graph is cleared from memory, the drawing area is cleared, the new graph is loaded into memory, and the new graph is displayed in the drawing area. The Save Graph button allows the user to save a graph that has been created for future use. This command provides a popup window asking the user to enter a name for the graph. These drop down menus and popup windows are also shown in Figure 3.7.

The box containing the Draw and Erase buttons is a toggle switch between drawing and erasing. The toggle is set to the button that is highlighted and can be switched by clicking the left mouse button with the cursor on the opposite button. When the user enters the environment, the toggle is set to Draw, so the user may immediately begin to draw the graph.

The huge white box is the drawing area where the user can draw the graph. Points can be drawn or erased by clicking the left mouse button. Points and edges are either drawn or erased depending on the setting of the toggle switch. The same mouse movements are used to draw and erase; so the result depends on the setting of the Draw and Erase toggle switch. When a point is drawn, it is given a number displayed in the center of a circle representing the graph point. The highest number represents the total number of points in the graph.
CHAPTER 3. A SIMPLE PROBLEM CLASS

PROBLEM
- All Pairs Shortest Path
- Connected Components

GRAPH TYPE
- Undirected
- Directed

proc_shell
- Number of Processors: 16
  1 16

OUTPUT
- Window
- File

dialogloadshell
- Enter file to be loaded:
  [Blank]
  Dialog Done

dialogsaveshell
- Enter file to be saved:
  [Blank]
  Dialog Done

Figure 3.7: Some of the menus for the xgraph environment.
CHAPTER 3. A SIMPLE PROBLEM CLASS

When a point is erased, all the edges connected to it are deleted, and the points numbered higher than this point are renumbered—that is, their number is decreased by one. This keeps the total number of points in the graph equal to the highest number of any point.

An edge can be drawn or erased between two points by pressing and holding the middle mouse button with the cursor on the origin point, moving the cursor to the destination point, and releasing the mouse button. The edge between these two points will then be drawn or erased, again depending on the setting of the Draw and Erase toggle switch. When a directed graph is chosen, the first point is the origin and the second is the destination, so the arrow is drawn pointing toward the second point. When the all pairs shortest path problem is being solved, after the user has drawn an edge, a popup menu appears prompting for the weight of this edge. After being specified the weight is displayed in a box at the midpoint of the drawn edge. The right mouse button is used to change the weight of an edge and is only allowed for the all pairs shortest path problem. The weight of an edge can be changed in the same manner that an edge is drawn and erased, that is, press and hold the right mouse button with the cursor on the origin point, move the cursor to the destination point, and release the mouse button.

Useful messages are displayed in the message box to indicate what action has been taken or what action is desired. It also indicates when a certain command is not executed and why. For instance, xgraph will not let the user draw points that are too close together. When the user tries to do this, a message saying “Point is too close to another point and cannot be drawn.” is displayed in the message window. The user always knows what the system is doing and can react appropriately.

This xgraph environment, as designed, is a problem solving environment for a fairly small class of problems, specifically the all pairs shortest path problem and the connected components problem for either a directed or an undirected graph. This is the type of tool that people need to solve graph problems on a parallel machine. An extension to make xgraph more of a parallel algorithm development environment would be to allow a user to specify the subroutine to be executed for each point or edge in the graph. The ability to
CHAPTER 3. A SIMPLE PROBLEM CLASS

specify this code could be added as another option for the type of problem. The environment would create the complete parallel program, and allow compiling, linking, and execution from within the environment. However, it is frequently the case that the user does not need an extremely general parallel algorithm development tool, but instead wants to leave the generic structure of the parallel algorithm fixed and concentrate on other aspects of the problem solving task. Problem specific environments seem much more flexible in handling these sorts of goals.

A limitation of the xgraph environment is that the size of the problem graph is constrained by the size of the window. The size of the problem graph is further constrained because points in the graph are not allowed to be placed too close together. Some additional features that could easily be added are clear screen, redraw, and undo. The clear screen button would remove the current graph from memory and clear the drawing area. The redraw button would clear the screen and redraw the current graph in memory on the drawing area. The undo button would allow the user to undo their most recent change to the graph. Some other features would include the ability to move a point once it has been drawn, and a choice to view the solutions as an adjacency matrix, or a list of pairs of connected points for the connected components problem and a list of pairs of points with an associated path length for the all pairs shortest path problem.

This xgraph environment allows us to specify an algorithm merely by selecting items from menus, but more importantly, it focuses on defining a specific problem. Next we describe how xgraph can be used to solve a specific problem.

3.3 Solution Using Application Specific Environment

This section describes the application of xgraph to a specific problem. A directed all pairs shortest path graph problem was selected to demonstrate the ease with which this environment can be used to successfully solve a particular problem. The all pairs shortest path problem is selected from the Problem menu. The directed graph is selected from the Graph
CHAPTER 3. A SIMPLE PROBLEM CLASS

Type menu. The number of processors is set by adjusting the slider with the middle mouse button.

After the algorithm has been specified, the user can define the problem by drawing the graph, using the left mouse button to draw the points and the middle mouse button to draw the edges as described earlier. After each edge is drawn the user is required to specify a weight for that edge.

Once the graph is drawn, the problem can be solved by clicking the left mouse button with the cursor on the Solve button. The solutions are displayed as an adjacency matrix in a window, as this is the default; however, the solutions can be easily saved in a file by clicking the left mouse button with the cursor on the Output button, selecting file, entering a file name, and re-solving the problem. The user can easily modify the graph and re-solve the problem. Figure 3.8 shows the xgraph environment displaying a sample problem, and Figure 3.9 shows the solution window.

3.4 Comparisons

Now that both environments have been used to solve the graph problems, comparisons are made regarding the specific use of each environment.

Each environment has a different primary focus, which visualizes different aspects of the problem. The BUILD environment focuses on the algorithm by allowing the user to express the algorithm in terms of a control flow graph, while the xgraph environment focuses on the problem definition by allowing the problem graph to be represented in terms of actual points and edges rather than a list.

The xgraph environment also helps the user specify the algorithm, which is implemented as a simple generic parallel algorithm expressed in the most natural form, namely, a few simple loops hardwired into the system. The algorithm parameters can be set by menu selections and slider settings, so the user does not have to deal with programming. If one person would build an application specific environment for a specific problem class and allow
Figure 3.8: Use of the xgraph environment to solve an all pairs shortest path problem.
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>0</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>8</td>
<td>15</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>15</td>
<td>0</td>
<td></td>
<td></td>
<td>3</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>13</td>
<td>0</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
<td>13</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>12</td>
<td>8</td>
<td></td>
<td></td>
<td>0</td>
<td>20</td>
<td>9</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10</td>
<td>14</td>
<td></td>
<td></td>
<td>6</td>
<td>0</td>
<td>7</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>3</td>
<td>25</td>
<td></td>
<td></td>
<td>17</td>
<td>11</td>
<td>0</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>6</td>
<td>2</td>
<td></td>
<td></td>
<td>5</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>8</td>
<td>9</td>
<td>18</td>
<td>19</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>9</td>
<td></td>
<td>7</td>
<td>0</td>
<td>13</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>15</td>
<td>3</td>
<td>12</td>
<td>13</td>
<td></td>
<td>7</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td>8</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td></td>
<td>11</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.9: Solution window for the sample problem shown in Figure 3.8.
CHAPTER 3. A SIMPLE PROBLEM CLASS

The user to set the important algorithm parameters by using menus, toggle buttons, sliders, etc., then many people could potentially reap the benefits by not having to waste their time programming a trivial parallel implementation. BUILD does provide help with some of the parallel problems discussed in Chapter 1; however, having to modify the generated code is a handicap to this environment because the user has to understand the programming primitives (in this case, the SCHEDULE package). This makes BUILD more suitable for a programmer interested in experimenting with parallel algorithms using the control flow paradigm.

The graph problem can actually be solved within the xgraph environment, whereas with BUILD the problem is not solved until the generated code has been modified, compiled, linked and executed outside of the BUILD environment. The algorithm parameters and the specific problem can easily be modified and re-solved in the xgraph environment, but making modifications when using BUILD could mean editing, recompiling, relinking, and rerunning. Hence, solving a problem is considerably more tedious in the BUILD environment than in the xgraph environment.

Overall, the xgraph environment is definitely more beneficial to the user seeking solutions. It provides a convenient way for the parallel algorithm to be specified, a specific problem to be defined, and solutions to be given within the environment.
Chapter 4

A Complex Problem Class

This chapter describes a more complicated example used to compare general purpose and problem specific environments. Section 4.1 describes how the problem is solved using the BUILD environment. Section 4.2 describes the application specific environment xpde that we built. Section 4.3 describes how the xpde environment can be used to solve a problem, and Section 4.4 compares the BUILD and xpde environments.

The complex problem is to find a numerical solution for a second order linear elliptic partial differential equation on the unit square. We consider only a simple model problem for the purposes of this work, namely, the Poisson problem $U_{xx} + U_{yy} = f$, where $f$ is a known function and $U$ is known on the boundary of the region. The generic algorithm for solving the problem is domain decomposition, which is based on dividing the original two dimensional region into smaller subregions, and solving the partial differential equation on the subregions (as independent subproblems) in parallel. Each subregion exchanges information with the neighboring subregions and recalculates its subproblem solution until a stopping condition is met. The whole process is fixed point iteration.

A high level description of the parallel algorithm is shown in Figure 4.1. The first parallel loop discretizes the problem by replacing the derivatives of the partial differential equation by finite difference formulas. The resulting linear equation is imposed at each interior grid

44
CHAPTER 4. A COMPLEX PROBLEM CLASS

Input: \( U_{xx} + U_{yy} = f \) where \( f \) is a known function and \( U \) is known on the boundary of the region

Output: approximation of \( U \) at grid points in the region

begin
  for \( i := 1 \) to MAX_SUBDOMAIN do in parallel
    discretization
  for \( i := 1 \) to MAX_SUBDOMAIN do in parallel
    form and factor preconditioner matrix
  while (not converged) do
    begin
      for \( i := 1 \) to MAX_SUBDOMAIN do in parallel
        apply the preconditioner
      for \( i := 1 \) to MAX_SUBDOMAIN do in parallel
        calculate matrix-vector multiplication
      final step to finish iteration
    end
end

Figure 4.1: Parallel algorithm for solving the partial differential equation.

point and the result is a linear system of equations \( Ax = b \). After this discrete system is formed, an iterative method is applied to find a solution. This matrix fixed point iteration can be accelerated with preconditioning. The preconditioner is a matrix \( M \) chosen so that \( AM^{-1}y = b \) (where \( y = Mx \)) is easier to solve than \( Ax = b \), in some sense. The domain decomposition algorithm is equivalent to solving \( Ax = b \) with a standard iterative technique and a particular preconditioning matrix \( M \). This latter conceptual viewpoint is the basis of the parallel algorithm. The second parallel loop initializes and factors this preconditioning matrix \( M \). The while loop handles the iterations by checking if the iteration has converged yet. By far the most expensive step in each iteration is the computation of \( AM^{-1}y^{(k)} \), where \( y^{(k)} \) is the current iterate. This is carried out in two parallel loops. The first parallel loop applies \( M^{-1} \) to the current iterate (by doing a forward and backward substitution with the factors of \( M \)) and the second parallel loop does a matrix-vector multiplication by \( A \). Notice that the two main steps of each iteration—applying the preconditioner and
CHAPTER 4. A COMPLEX PROBLEM CLASS

matrix-vector multiplication—can be done in parallel, on a subdomain by subdomain basis. Both the preconditioning matrix $M$ and the original matrix $A$ can be defined in such a way that they are almost completely made up of independent blocks associated with each of the subdomains. The final step of the iteration consists of a few sequential computations that must be performed during each iteration (e.g., updating certain iteration parameters, computing the convergence criteria). The pseudocode in Figure 4.1 shows the general structure of the algorithm; some details vary depending on the specific iterative method and preconditioner used. The solution to the linear system gives the desired approximations to the unknown function $U$.

The algorithm for domain decomposition that we are using is for a shared memory machine, which makes exchanging information with neighboring subregions straightforward. An algorithm using domain decomposition on a distributed memory system would have to distribute the matrices $A$ and $M$ representing the preconditioned linear system to the processors, and also have to deal with exchanging information between processors.

4.1 Solution Using BUILD

This section describes how BUILD can be used to solve this complex problem, specifying the parallel algorithm to BUILD by drawing a control flow graph. In this situation, we want to solve each of the subproblems in parallel, so the algorithm for solving a Poisson problem using domain decomposition involves setting up the discretization matrix, setting up the preconditioning matrix, solving the subproblem on each subregion (by applying the preconditioner and doing a matrix-vector multiplication), and repeating the solution of subproblems until a convergence criterion is met.

The control flow graph is specified by using loop nodes (specifying $n$ computations) when an iteration is necessary. BUILD does not allow one to iterate until a condition is satisfied, thus, the while loop iteration must be added by the programmer. Hence, we separate the algorithm into two parts, the initialization steps and the iteration step, to isolate the code.
CHAPTER 4. A COMPLEX PROBLEM CLASS

that would be inside the conditional iteration. The dependency graph for each of these
steps is drawn separately with BUILD. The programmer has to modify the code outside of
BUILD to create the conditional iteration as well as combine the generated code for each
of the steps.

The initialization steps include initializing the discretization matrix and initializing and
factoring the preconditioning matrix. A dependency graph for this step would be a graph
with two loop nodes and an arc between the two loop nodes to illustrate the dependency.
The iteration step includes applying the preconditioner, calculating the matrix-vector mul-
tiplication, and the final step. The first two (iteration) steps are loop nodes and the last
step is a regular node. Figure 4.2 illustrates the control flow graph of a single iteration step.
Note that BUILD does not show any physical distinction between regular nodes and loop
nodes in the graph. The top loop node in the graph corresponds to the application of the
preconditioner. The second loop node is drawn for doing the matrix-vector multiplication.
The last node is a regular node to calculate the final step of each iteration. Dependencies
exist between each of these steps, so arcs must be drawn. This graph does not illustrate
the fact that the calculations need to be repeated; the programmer must encode this outer
loop when modifying the generated code.

A subroutine needs to be specified for each node in the control flow graph. As indicated
in Chapter 3, BUILD provides a way to specify and modify the code for the subroutine
of a particular node. Each of the nodes in the control flow graph in Figure 4.2 executes a
different subroutine. After specifying and assigning a subroutine for each node, the code
can be generated and the graph saved. This must be done for both the initialization steps
and the iteration step.

BUILD has fulfilled its purpose by generating code with the SCHEDULE calls which
handle the spawning, mapping, and synchronizing of tasks. We have two separate pieces
of code, one for the initialization step and the other for the iteration step, that have to
be pieced together and modified to create an executable routine. We describe how the
generated code of the iteration step (shown in Figure 4.3) is transformed into a conditional
Figure 4.2: Use of BUILD to solve partial differential equation problem.
CHAPTER 4. A COMPLEX PROBLEM CLASS

iteration.

As in the example in Chapter 3, making the generated code executable entails adding variable declarations, initializations, subroutine parameters, and possibly output statements. It is necessary to declare the variables used. Some variables may also have to be placed in COMMON blocks due to the nature of SCHEDULE. Since this is a more complex problem, more COMMON blocks are used. BUILD does not provide a way to select the number of processors, nproc, so it must be initialized or programmed to be defined at run-time. This initialization is part of the initialization step. Using the loop node means that it is necessary to initialize the number of iterations of each of the loops. It is also possible to define the number of loop iterations at run-time, but the code to make this possible must be added. The parameters for subroutines have to be added for the subroutines the programmer defined. In addition to these changes, the programmer must create a conditional iteration. The modified code for the iteration step is shown in Figure 4.4 with changes in italics. The generated code is converted into a subroutine for a single iteration of the conditional iteration step. This subroutine can then be called from within a while loop that controls the conditional iteration. Notice that the main SCHEDULE routine sched will be called once per iteration with this approach. Once the generated code of the initialization step is modified and the iteration step is integrated with it, the program can be compiled, linked, and executed.

BUILD provides a way to spawn, map, and synchronize tasks. This domain decomposition algorithm calls for repetition of tasks until a certain condition is met. This proved to be a difficult maneuver using SCHEDULE, and modifying the code to achieve this repetition required a deep understanding of the SCHEDULE package. Now let us consider what an application specific environment looks like to solve this Poisson problem.
external paralg
call sched(nproc,paralg)
stop
end
subroutine paralg
external loop1, loop2, finish_iter
integer jobtag, icango, nchks, mychks(1), numits
jobtag = 1
icango = 0
nchks = 1
mychks(1) = 2
call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, loop1, jobtag, numits)
jobtag = 2
icango = 1
nchks = 1
mychks(1) = 3
call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, loop2, jobtag, numits)
jobtag = 3
icango = 1
nchks = 0
call dep(jobtag, icango, nchks, mychks)
call putq(jobtag, finish_iter)
return
end
subroutine loop1(myid,n)
integer myid, n
logical wait
external precond_apply
go to (1100,1200), ientry(myid,2)
1100 continue
do 1150 i = 2, n
call nxtag(myid, jdummy)
call spawn(myid, jdummy, precond_apply)
1150 continue
call precond_apply
if (wait(myid,2)) return
1200 continue
return
end
subroutine loop2(myid,n)
integer myid, n
logical wait
external mat_mult
go to (2100,2200), ientry(myid,2)
2100 continue
do 2150 i = 2, n
call nxtag(myid, jdummy)
call spawn(myid, jdummy, mat_mult)
2150 continue
call mat_mult
if (wait(myid,2)) return
2200 continue
return
end
subroutine do_iter
  external paralg
  call sched(nproc,paralg)
  stop
end

subroutine paralg
  external loop1, loop2, finish_iter
  integer jobtag, icango, nchks, mychks(1), numits
  COMMON / c1ivdd / nsubx, nsuby, nx, ny
  numits = nsubx*nsuby
  jobtag = 1
  icango = 0
  nchks = 1
  mychks(1) = 2
  call dep(jobtag, icango, nchks, mychks)
  call putq(jobtag, loop1, jobtag, numits)
  jobtag = 2
  icango = 1
  nchks = 1
  mychks(1) = 3
  call dep(jobtag, icango, nchks, mychks)
  call putq(jobtag, loop2, jobtag, numits)
  jobtag = 3
  icango = 1
  nchks = 0
  call dep(jobtag, icango, nchks, mychks)
  call putq(jobtag, finish_iter)
  return
end

subroutine loop1(myid,n)
  COMMON / c1coef / r1coef(1)
  COMMON / c1idco / i1idco(1)
  COMMON / c1ivdi / i1neq, i1mneq, i1ncoe, i1mnco
  COMMON / c1endx / i1endx(1)
  COMMON / c1undx / i1undx(1)
  COMMON / c1ivdd / nsubx, nsuby, nx, ny
  COMMON / c1feq0 / firsteqn
  integer firsteqn
  COMMON / c1feqs / firsteqn_s
  integer firsteqn_s
  COMMON / c1feqw / firsteqn_w
  integer firsteqn_w
  COMMON / c1abd0 / abd_interior
  COMMON / c1abds / abd_interface_s

Figure 4.4: Modified code generated by BUILD to solve the iteration step.
CHAPTER 4. A COMPLEX PROBLEM CLASS

```
COMMON / cflagdw / abd_interface_w
COMMON / ciabde / abd_crosspts
COMMON / clunkn / y
integer myid, n
logical wait
external precond_apply
go to (1100, 1200), ientry(myid, 2)
1100 continue
    do 1150 i = 2, n
        call nxtag(myid, jdummy)
        call spawn(myid, jdummy, precond_apply,
            + nszb, nsuby, nx, ny, r1coef, ilidco, ilmneq, ilneqn, ilendx,
            + ilundx, firsteqn, firsteqn_s, firsteqn_w, abd_interior,
            + abd_interface_s, abd_interface_w, abd_crosspts, y)
    1150 continue
    call precond_apply
    + (nszb, nsuby, nx, ny, r1coef, ilidco, ilmneq, ilneqn, ilendx,
        + ilundx, firsteqn, firsteqn_s, firsteqn_w, abd_interior,
        + abd_interface_s, abd_interface_w, abd_crosspts, y)
    if (wait(myid, 2)) return
1200 continue
    return
end subroutine loop2(myid, n)
COMMON / clcoef / r1coef(1)
COMMON / clidco / ilidco(1)
COMMON / clvdi / ilneqn, ilmneq, ilncoe, ilmnc0
COMMON / cliendz / ilendx(1)
COMMON / clundx / ilundx(1)
COMMON / clunkn / y
COMMON / c1bbbb / x
integer myid, n
logical wait
external mat_mult
go to (2100, 2200), ientry(myid, 2)
2100 continue
    do 2150 i = 2, n
        call nxtag(myid, jdummy)
        call spawn(myid, jdummy, mat_mult,
            + z, y, r1coef, ilidco, ilmneq, ilncoe, ilendx, ilundx)
    2150 continue
    call mat_mult
    + (z, y, r1coef, ilidco, ilmneq, ilncoe, ilendx, ilundx)
    if (wait(myid, 2)) return
2200 continue
    return
end
```

Figure 4.4 (con't): Modified code generated by BUILD to solve the iteration step.
4.2 Description of the Application Specific Environment

The application specific environment xpde used to solve a Poisson problem via domain decomposition was also built using the Athena widgets provided in the X Window System. An application specific environment provides a convenient method for specifying the algorithm and the specific problem to be solved. The goal is to allow the user to experiment with several versions of a parallel domain decomposition algorithm for numerically solving a partial differential equation. In particular, xpde is designed to allow the user to experiment with different combinations of methods (discretization, iterative solver, preconditioner), decompositions (number of subdomains, assignment to processors, etc.), and Poisson problems. We begin this section by identifying the important characteristics of this problem that an environment needs to address.

An environment developed for a specific application provides a convenient method for setting algorithm parameters and defining the specific problem. The algorithm used in an application specific environment is a generic parallel algorithm built into the system. In this situation, the specific instance of the domain decomposition algorithm may be specified by selecting a discretization method, choosing a linear equation solver (which consists of selecting a basic iterative method and a preconditioner), and setting the convergence criterion. The algorithm is also defined in problem specific terms by choosing the number of processors, decomposing the domain, defining a grid, assigning subdomains to processors, and selecting the desired output data. The specific problem can be defined by providing the following functions: $f$ where $U_{xx} + U_{yy} = f$ on the unit square, $g$ where $U = g$ at the boundary, and true for the actual values (true is often the same as $g$) (Note that true would not be known in advance for "real" problems, but $f$ and $g$ are often chosen so that true is known in a research setting.)

The types of discretization methods that may be used include five point star [Rice 85] and HODIE (High Order Difference approximation with Identity Expansion) [Rice 85]. The five point star method is a standard second order accurate centered finite difference method.
CHAPTER 4. A COMPLEX PROBLEM CLASS

The HODIE method is a high order accurate (up to sixth order) finite difference method. The types of iterative methods include classical conjugate gradient [Hestenes 52] and GMRES [Saad 86], a generalized minimum residual method. The preconditioner may be either tangential [Chan 90] or probe [Chan 90]. Both preconditioning methods choose as a preconditioner \( M \) a block upper triangular approximation to \( A \). They differ in how the equations associated with unknowns on subdomain boundaries are approximated.

The number of processors and the convergence criterion are important characteristics that can be set by a user. The domain is decomposed by specifying the number of subdomains in the \( x \) direction and the number of subdomains in the \( y \) direction. Each subdomain is further decomposed by a grid, which is defined by specifying the number of grid lines in the \( x \) direction and the number of grid lines in the \( y \) direction. The environment must also provide a way to assign subdomains to processors.

The graphical user interface of xpde (shown in Figure 4.5) allows the user to solve a Poisson equation by using domain decomposition, specifying parameters for the algorithm and problem by drop down menus (shown in Figure 4.6) and popup windows (shown in Figure 4.7). The xpde environment allows the user to specify the domain decomposition method by choosing a discretization method, a linear equation solver, and a preconditioner. The Discretization button is a drop down menu that allows the user to choose either the Five Point Star or the HODIE method. The Iterative Solver button is a drop down menu that allows the user to choose either the Conjugate Gradient or the GMRES method. The Preconditioner button is also a drop down menu that allows the user to select either the Tangential or Probe method. The environment is set up with default settings, which are the Five Point Star, the Conjugate Gradient, and the Tangential methods. The Convergence Criterion button is also a popup window with a slider. The user can set the convergence criterion by adjusting the slider to the desired setting with the middle mouse button. The convergence criterion may vary in powers of ten from \( 10^{-10} \) to \( 10^{-1} \). The default setting for the convergence criterion is \( 10^{-10} \).

The algorithm is also specified by the number of processors, decomposition, and con-
Figure 4.5: The application specific partial differential equation environment—xpde.
### Chapter 4. A Complex Problem Class

**Discretization**
- Five Point Star
- HODIE

**Iterative Solver**
- Conjugate Gradient
- GMRES

**Preconditioner**
- Tangential
- Probe

**Mapping**
- Natural
- Block-Row
- Block-Column
- Block-Block

**Problem Functions**
- Specify Filenames
- Specify Strings
- Use Default Functions

---

Figure 4.6: The drop down menus for the xpde environment.

Vergence criterion. The *Number of Processors* button is a popup window that enables the user to set the number of processors by adjusting the slider to the desired setting with the middle mouse button. The *Decomposition* button is a popup window that allows the user to decompose the domain by setting the number of subdomains in both the $x$ and $y$ directions and in each subdomain the number of grid lines in both the $x$ and $y$ directions. These values are all set by adjusting a slider to the desired setting with the middle mouse button. The default settings for each of these characteristics is one.

The subdomains can be assigned to processors by using the *Mapping* button. Before assigning the mapping, the number of processors and the number of subdomains in the $x$ and $y$ directions should be specified, since the mapping algorithms need this information.
Figure 4.7: The popup windows for the xpde environment.
CHAPTER 4. A COMPLEX PROBLEM CLASS

to make the assignments. The Mapping button is a drop down menu with four possible mapping algorithms: Natural, Block-Row, Block-Column, Block-Block. Each mapping assigns subdomains to the number of processors specified earlier. The Natural mapping incrementally assigns each subdomain to a processor starting with zero at the bottom left subdomain. The Block-Row mapping assigns each row of subdomains to a processor, starting with zero at the bottom row of subdomains. The Block-Column mapping assigns each column of subdomains to a processor, starting with zero at the left column of subdomains. These three algorithms cycle back to the processor zero and continue mapping until all subdomains are assigned. The Block-Block mapping assigns a block of subdomains to a processor, which is not possible for all decompositions, because the number of subdomains assigned to a processor must be a perfect square. The Mapping button is selected to assign subdomains to processors. There is no default setting because a mapping must be selected in order to assign subdomains to processors. Each of these mapping techniques is illustrated in Figure 4.8.

The Poisson problem to be solved can be defined in several ways, but must be defined. The Problem Functions button is a drop down menu that allows the user to Specify Filenames, Specify Strings, or Use Default Functions. If the user chooses Specify Filenames, a popup window appears prompting for files that contain FORTRAN functions for \( f \), \( g \), and \( \text{true} \) (We assume the FORTRAN default type declarations for variables). In order for the specification to take effect the user must click the left mouse button with the cursor on the DONE button. If the user chooses Specify Strings, a popup window appears prompting for strings that describe the FORTRAN functions for \( f \), \( g \), and \( \text{true} \). Again in order for the specifications to take effect the user must click the left mouse button with the cursor on the DONE button. If the user chooses Use Default Settings, a popup window appears illustrating the default functions for \( f \), \( g \), and \( \text{true} \). Once again the user must click the left mouse button with the cursor on the DONE button for the defaults to take effect.

The Output Data button is a popup window providing toggle buttons for each type of output data that the user may select to display when the problem is solved. The types of
Figure 4.8: The four mapping techniques of the xpde environment.
CHAPTER 4. A COMPLEX PROBLEM CLASS

output data includes: Solution Values at Grid Points, Max Solution Value at Grid Points, Error Values at Grid Points, Max Error Value at Grid Points, and Execution Time. The default settings for the output data are Max Error Value at Grid Points and Execution Time.

The Solve button solves the Poisson problem with the algorithm determined by the settings that have been specified. The output solutions specified by the settings of the toggle buttons in the Output Data popup window are displayed in a popup window, which provides scrolling in both directions when necessary. When scrolling down or right click the left mouse button, when scrolling up or left click the right mouse button, and when repositioning click the middle mouse button. An additional feature would be the capability to illustrate the solutions as a contour or surface plot.

The Help button was added to provide help to a novice user. The help information is displayed in a popup window providing scrolling. The Quit button does the expected and exits from the application environment. The long narrow box is used to display helpful messages during the execution of the environment. Useful messages are displayed to indicate what the environment is doing.

Since most of the algorithm parameters are set with drop down menus, the user may forget what the current settings are. The Show Current Settings is a popup window that displays the current settings for all the variable algorithm characteristics. The Show Processor Mapping draws a chart illustrating what subdomains are assigned to what processors. Each of the processors is shown in a different color, which makes it easy to distinguish which processors are assigned which subdomains. Some sample mapping charts are shown in Figure 4.8.

This environment is specifically designed for solving Poisson’s equation via domain decomposition. It provides a convenient way to specify the algorithm and the problem by allowing many characteristics to be manipulated within the environment. Next we describe how this problem specific environment can be used to solve a specific problem.
CHAPTER 4. A COMPLEX PROBLEM CLASS

4.3 Solution Using Application Specific Environment

The xpde environment was designed to allow experimentation with different methods for solving a partial differential equation using domain decomposition. The algorithm and problem characteristics may be selected in any order with one condition: the number of processors and the number of subdomains in both the $x$ and $y$ directions need to be specified before the mapping can be assigned.

We can specify the algorithm and problem by selecting various items from menus, adjusting sliders to the appropriate values, and entering keyboard information to define the Poisson problem to be solved. The number of processors is set by clicking with the cursor on the *Number of Processors* button and adjusting the slider to the value four. The decomposition can be specified by clicking with the cursor on the *Decomposition* button and adjusting the sliders for the number of subdomains in the $x$ and $y$ directions to each be four and the number of $x$ and $y$ grid lines per subdomain to each be four. The convergence criterion is set by clicking with the cursor on the *Convergence Criterion* button and adjusting the slider to the value $10^{-5}$.

Since the number of processors and the domain decomposition have been specified, mapping of the subdomains to processors can be done. This is done by selecting the *Block-Block* mapping algorithms from the *Mapping* menu. To see the mapping distribution click with the cursor on the *Show Processor Mapping*. Next, select the *Five Point Star* discretization method from the *Discretization* menu, select the *GMRES* iterative method from the *Iterative Solver* menu, and select the *Tangential* preconditioning method from the *Preconditioner* menu.

Next, we need to define the problem functions by clicking with the cursor on the *Problem Function* button. We elect to *Use Default Functions*, so after this popup window appears we click with the cursor on the *DONE* button. We use the default output settings of *Max Error Value at Grid Points* and *Execution Time*. Finally, the problem can be solved by clicking with the cursor on the *Solve* button. The solutions that we indicated are displayed in a
CHAPTER 4. A COMPLEX PROBLEM CLASS

popup window (shown in Figure 4.9). The algorithm and problem characteristics described can easily be changed and the problem re-solved. In this way, the user can experiment with the effects of using a different mapping, more or less processors, more or less subdomains, more or less grid points per subdomain, a different convergence criterion, different output data and/or a different problem. The user may also experiment with different discretization methods, iterative solvers, and preconditioners.

4.4 Comparisons

Now that both environments have been used to solve the partial differential equation problem, comparisons can be drawn, regarding the specific use of each environment.

Both environments focus on the algorithm, but xpde also focuses on the problem. BUILD's approach is very low level, and the xpde approach is very high level. The BUILD environment focuses on visualizing the algorithm by expressing it in terms of a control flow graph. The xpde environment focuses on experimenting with various algorithm characteristics and Poisson problems, and provides a visualization of the mapping of subdomains to processors. Some of the characteristics that can be experimented with only deal with specifying the algorithm, such as the discretization methods, iterative solvers, preconditioners, and the convergence criterion, while others only deal with defining the specific problem, such as defining the problem functions. The remaining characteristics—number of processors, domain decomposition, grid definition, mapping of subdomains to processors, and output data—also experiment with the algorithm, but in very problem specific terms. The xpde environment implements a generic algorithm that depends on menu selections, slider settings, and keyboard input.

The xpde environment is designed for the user who wants to experiment with various aspects of solving a Poisson equation via domain decomposition. Combinations of the various algorithm and problem characteristics can be altered and then the Poisson equation re-solved, or a different Poisson equation can be solved. This environment was designed
**CHAPTER 4. A COMPLEX PROBLEM CLASS**

---

**discretization module**

---

**5-point star**

<table>
<thead>
<tr>
<th>domain</th>
<th>rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>discretization</td>
<td>uniform</td>
</tr>
<tr>
<td>number of equations</td>
<td>121</td>
</tr>
<tr>
<td>max no. of unknowns per eq.</td>
<td>5</td>
</tr>
<tr>
<td>matrix is</td>
<td>non-symmetric</td>
</tr>
</tbody>
</table>

**execution successful**

<table>
<thead>
<tr>
<th>its</th>
<th>res. norm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.49892636E+03</td>
</tr>
<tr>
<td>1</td>
<td>0.13631790E+03</td>
</tr>
<tr>
<td>2</td>
<td>0.68596066E+02</td>
</tr>
<tr>
<td>3</td>
<td>0.17231141E+02</td>
</tr>
<tr>
<td>4</td>
<td>0.90080355E+01</td>
</tr>
<tr>
<td>5</td>
<td>0.36091767E+01</td>
</tr>
<tr>
<td>6</td>
<td>0.12594197E+01</td>
</tr>
<tr>
<td>7</td>
<td>0.76042403E+00</td>
</tr>
<tr>
<td>8</td>
<td>0.26679520E+00</td>
</tr>
<tr>
<td>9</td>
<td>0.13881839E+00</td>
</tr>
<tr>
<td>10</td>
<td>0.64391741E-01</td>
</tr>
<tr>
<td>11</td>
<td>0.21008164E-01</td>
</tr>
<tr>
<td>12</td>
<td>0.76738911E-02</td>
</tr>
<tr>
<td>13</td>
<td>0.27425849E-02</td>
</tr>
<tr>
<td>14</td>
<td>0.60854015E-03</td>
</tr>
</tbody>
</table>

---

**ellpack output**

```
+ max( abs(error ) ) on 13 x 13 grid = 3.6173929E-05  
+ 11 norm( error ) on 13 x 13 grid = 1.3575316E-05  
+ 12 norm( error ) on 13 x 13 grid = 1.7954918E-05  
```

**total time** 0.97

---

Figure 4.9: Solution window for the Poisson problem described in Section 4.3.
CHAPTER 4. A COMPLEX PROBLEM CLASS

to allow a user to experiment with different characteristics of the algorithm and different Poisson problems. For instance, the user can change the specific problem to be solved by entering new problem functions; the user can change the algorithm in problem specific terms by selecting a new domain decomposition or selecting a different mapping; or the user can change the algorithm by selecting a different discretization method, iterative solver, preconditioner, or convergence criterion. This application specific environment is geared toward experimentation. It does not involve any programming from the user, because a generic form of the parallel algorithm is hardwired into the system allowing algorithm and problem characteristics to be altered for experimentation.

The BUILD environment is designed for a programmer who is required to develop a parallel algorithm using the control flow paradigm. The solution of a partial differential equation solved via domain decomposition was difficult to implement in BUILD, because it does not have a construct that allows a conditional iteration. This caused the programmer to add this condition around the code that was output by BUILD. Using BUILD required the programmer to understand the SCHEDULE package. Modifying the code generated by BUILD is a definite handicap to using BUILD for a problem of this type.

A problem can actually be solved within the xpde environment. The characteristics of the algorithm and the problem can be modified by using menus, sliders and keyboard input. An experimental environment of this type provides an easy way to make modifications by using a menu- and mouse-oriented graphical user interface. This allows a problem to be easily altered and re-solved with the solutions displayed immediately in a popup window. However, getting solutions using BUILD is not as straightforward. The user must modify the generated code, compile it, link it, and run it outside of the BUILD environment before getting solutions. Any modification to the problem or solution method using BUILD generally means editing, recompiling, relinking, and rerunning.

Once again, if users are after solutions to specific problems and want to experiment conveniently with a different algorithm or problem, an application specific environment seems more appropriate. It permits them to specify characteristics of the algorithm and
CHAPTER 4. A COMPLEX PROBLEM CLASS

problem by selecting items from a menu, adjusting the values of sliders, or entering data using the keyboard. As indicated in Chapter 3, the BUILD environment would be more appropriate for a programmer who is interested in solving various parallel problems using the control flow approach. The xpde environment is designed specifically for the users who want to experiment with different methods, decompositions, and Poisson problems.
Chapter 5

Conclusions

In this chapter we discuss the conclusions that our case study comparison supports. Then we compare general purpose and problem specific parallel programming environments along the lines of several important software quality attributes. Finally, we answer the questions posed in Chapter 1.

Our work illustrates that general purpose environments have serious drawbacks. They force the user to concentrate on mapping the problem to the environment, because these environments are based on a single abstract paradigm that is often inconvenient for the problem at hand. Furthermore, the user often must have knowledge of the underlying parallel programming primitives used to describe the parallel algorithm. General purpose environments do not provide a means for the user to define a specific problem or the ability to experiment easily with different versions of the algorithm or problem. They just provide an environment for developing a parallel algorithm from scratch. The primary interest of a user is not always the development of the parallel algorithm. The user is often interested in other aspects of the problem solving task.

Problem specific environments, on the other hand, have several distinct advantages. The key advantages are the ability to specify the algorithm as well as define the problem within the environment. The user is provided with an environment where one can experiment
CHAPTER 5. CONCLUSIONS

with different problem parameters and different problems. One can obtain solutions and make modifications within the environment. Another advantage is that the parallelism of the problem is represented in a more natural, problem specific way. For instance, the user can concentrate on solving the partial differential equation by thinking about the division of the region and the assignment of the resulting subdomains to processors. This process is further enhanced by providing a visualization of this mapping. Overall, problem specific environments increase usability and productivity.

Besides the fact that problem specific environments are much more attractive than general purpose environments from a user's point of view, our work suggests that problem specific environments really are not any harder to build than general purpose environments are to use. In fact, given someone with programming experience with a graphical display system (or someone who knows how to build an interface with an interface builder) and knowledge about parallel programming, an efficient problem specific environment can be created as efficiently as using a general purpose environment. One of the reasons is because of the disadvantages of general purpose environments previously discussed. One already has to know how to develop the parallel program anyway, so developing it for the problem specific environment should not be more difficult than developing it in the general purpose environment.

We now compare the general purpose environments and problem specific environments along the lines of several important software quality attributes [Sommer 89].

Economy. Economy refers to whether the development of the system will be cost-effective. The development of general purpose environments would save programmers from having to deal with some of the parallel programming problems mentioned in Chapter 1. Although the programmer is relieved of some parallel programming problems, the programmer has to concentrate on mapping the problem to the environment. The development of a problem specific environment would mean only one person (or group) had to develop the application and many users could reap the benefits.
CHAPTER 5. CONCLUSIONS

Documentation. Documentation refers to design specifications, comments in the code, user manuals, help facility, etc. Most general purpose environments are currently in the research stage, so the documentation for using these systems is typically just reports, theses, or dissertations, which are generally not geared toward an average user. In order for these tools to be used, detailed documentation must be provided. Both types of environments are on an equal level as far as documentation is concerned.

Efficiency. Both types of environments are fairly efficient in terms of run-time execution, although there may be cases where the general purpose environment forces the user to accept some unnecessary overhead (e.g., calling sched every iteration to redefine the same dependency graph). In terms of development time; however, problem specific environments are more efficient, because the problem can be solved while inside the environment after specifying certain algorithmic characteristics and defining the specific problem. The general purpose environments require defining the algorithm within the environment, in addition to compiling, linking, and running the program outside of the environment.

Flexibility. The general purpose environments are based on a single abstract paradigm, so the programmer is forced to map a problem to this paradigm. This seems to imply that general purpose environments are not flexible, but they do provide the flexibility to solve many different problems. The application specific environments are not flexible from problem class to problem class, but it is very easy to alter the problem parameters for a given problem class. Hence, both types of environments are flexible, but in a different sense.

Functionality. Environments that allow one to save and reload stored information have a definite advantage over those that do not. This is especially true for general purpose environments, because no one wants to solve the same problem again. For example, BUILD allows one to save the control flow graph. Therefore, if at a later point in time one wants to modify the control flow graph for a specific problem, it will not
CHAPTER 5. CONCLUSIONS

be necessary to redraw the whole graph. It may be the case that some application
specific environments will not require any information to be saved. For example, in
the xpde environment it is not particularly advantageous to save any information,
whereas in the xgraph environment it is advantageous to save the graph for possible
future reference.

Interoperability. Interoperability refers to whether the system can be divided and exe-
cuted on more than one system. This depends on the networking capabilities of the
computer system being used. These environments consist of two parts — the back
end which does the computation and the front end which presents the graphical user
interface. If interoperability is achieved, the environment can be executed on one
machine and displayed on another. For example, the X Window System helps achieve
this through the client/server division. Both types of environments are on an equal
level as far as interoperability is concerned.

Maintainability. Both types of environments need to be easily maintainable to be effec-
tive. When changes need to be made to fix bugs, add features, etc., the changes
should not be more trouble than they are worth. Therefore, an environment must be
upgradable. Both types of environments are on an equal level as far as maintainability
is concerned.

Portability. Environments should be developed so that they can be moved freely to differ-
ent machines, especially since hardware technology is changing quickly. We achieve a
degree of portability by basing our application specific environments on the X Window
System, because the X Window System runs on a variety of computers and worksta-
tions from many different vendors. Ideally, it would be convenient if environments
did not depend on shared memory or distributed memory machines. Since these ar-
chitectures are extremely different, problems must be solved in different ways. Both
types of environments are about equally portable.
CHAPTER 5. CONCLUSIONS

Productivity. The use of general purpose environments should increase productivity, because programmers do not have to deal with all the detail of the parallel programming primitives for a particular parallel machine. Nevertheless, they would have to figure out how to map their specific problem to the paradigm used in this tool. The use of application specific environments would definitely increase productivity. The generic parallel algorithm is already known to the system. The solution to the problem can be obtained immediately by using the mouse- and menu-oriented graphical user interface to specify certain algorithmic characteristics, define the specific problem, and solve the problem.

Reliability. Both types of environments must be well tested. Programmers using general purpose tools do not want their system to fail when they are in the middle of creating a parallel program. The scientists and engineers using the problem specific environments depend on receiving correct answers without failures.

Understandability. Understandability refers to how easy the system is to understand. Both types of environments provide a mouse- and menu-oriented graphical user interface to aid understanding. A help facility also improves understandability.

Usability. Usability refers to how easy the system is to use and to learn. Both types of environments enhance usability by using a mouse- and menu-oriented graphical user interface. Nevertheless, the application specific environment is better, because it can be used by both novices and experts to solve specific problems. The current general purpose environments still require an understanding of the details of parallel programming, which limits the use of these environments.

Although this is not a complete list of software quality attributes, we feel that these are the most important. The development of software tools for parallel programming must consider these attributes. The challenge of providing software support for parallel computing is that the scientists and engineers, who need parallel processing to achieve a desired level of
CHAPTER 5. CONCLUSIONS

performance, may not have the computer science background (or possibly the time) to learn all the details of every parallel machine they may use. The high-performance computing community needs software tools to enable them to complete their work in a timely manner, which is the province of computer scientists.

One of the motivating questions we considered is whether or not programming tools actually reflect the way people think about and develop their parallel programs. In Chapter 2 we discussed several different approaches for the development of parallel programs represented by current general tools. These tools are based on a single abstract paradigm that force the user to concentrate on mapping their problem to the environment. We feel that an application specific tool would better reflect the way people think about a parallel program, because the application specific tool would be designed to reflect the parallelism in a more natural way.

In regards to the question asked in Chapter 1 about what special purpose parallel programming environments would look like, we feel the environments should have a graphical user interface and a hardwired generic form of a parallel algorithm built into the system. The user can specify the algorithm and define the problem by menu selections, slider settings, toggle selections, keyboard input, etc. Solutions (and possibly performance data) can easily be given within the environment. Modifications to the algorithm or problem definition can also be done within the environment.

We feel the advantages application specific environments provide over general purpose environments are sufficient to warrant the development of application specific environments. One of the major advantages is that application specific environments allow the user to concentrate on the problem definition, as well as the algorithm. The parallelism of the problem is also represented in a more natural way. They require little (if any) parallel programming and give solutions and allow modifications within the environment. However, they are limited to a specific problem, which is a disadvantage, because general purpose environments can be used for many different problems. The developers of application specific environments have to develop a parallel program, as well as have some expertise in a graphical
CHAPTER 5. CONCLUSIONS

display system, such as the X Window System. Application specific environments for parallel computing are a form of the software support for parallel computing that is needed by scientists and engineers.

This software support can come in many varieties, although our main interest in this thesis has been programming environments. The use of general purpose environments still requires a vast amount of knowledge about parallel programming primitives and paradigms, as well as the parallel machine. Ideally, it would be beneficial if a general purpose environment could be adaptable, so that an application specific tool could be developed from it by tuning and modifying it. An application specific environment consists of two parts, a graphical user interface and a generic form of a parallel algorithm to solve a specific problem. The general purpose environment currently helps the user specify the algorithm, however, it would also need to allow an interface to be built. This means that the adaptable general purpose environment would need to contain an interface builder, as well as constructs to aid parallel programming. This adaptable general purpose environment would make the development of an application specific environment easier and also provide valuable support for parallel programming. Therefore, future work includes creating an adaptable general purpose parallel programming environment.
REFERENCES


REFERENCES


REFERENCES


REFERENCES


76
VITA

Loretta Sue Auvil was born on March 14, 1968 to Clyde and Gladys Auvil. She calls home—Philippi, West Virginia. She graduated in 1986 from Philip Barbour High School. She continued her education at Alderson-Broaddus College in Philippi, West Virginia, graduating in 1990 with a Bachelor of Science in Applied Mathematics and Computer Science. During her senior year, she participated in the Science and Engineering Research Semester Program at Oak Ridge National Laboratory. During her work at Oak Ridge, she was inspired to further her education. So she began to look for ways to fund this advanced degree and places to receive this degree. She was accepted into the United States Air Force Palace Knight Program and she chose to pursue her Master’s degree in Computer Science at Virginia Polytechnic Institute and State University. After completing her Master’s degree, Loretta will be working as a civilian for the United States Air Force at Rome Laboratory, Griffiss Air Force Base in Rome, New York.

Loretta Sue Auvil

77