

AN ABSTRACT OF THE THESIS OF

Damien D. Macielinski for the degree of Master of Science in Computer Science

presented on October 28, 1994.

Title: Task-Parallel Extension of a Data-Parallel Language.

Redacted for Privacy¹

Abstract Approved: _____

Dr. Walter G. Rudd

Two prevalent models of parallel programming are data parallelism and task parallelism. Data parallelism is the simultaneous application of a single operation to a data set. This model fits best with regular computations. Task parallelism is the simultaneous application of possibly different operations to possibly different data sets. This fits best with irregular computations. Efficient solution of some problems require both regular and irregular computations. Implementing efficient and portable parallel solutions to these problems requires a high-level language that can accommodate both task and data parallelism. We have extended the data-parallel language Dataparallel C to include task parallelism so that programmers may now use data and task parallelism within the same program. The extension permits the nesting of data-parallel constructs inside a task-parallel framework. We present a banded linear system to analyze the benefits of our language extensions.

© Copyright by Damien D. Macielinski

October 28, 1994

All Rights Reserved

Task-Parallel Extension of a Data-Parallel Language

by

Damien D. Macielinski

A THESIS

submitted to

Oregon State University

**in partial fulfillment of the
requirements for the degree of**

Master of Science

Completed October 28, 1994

Commencement June 1995

Master of Science thesis of Damien D. Macielinski
presented on October 28, 1994

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Chair of Computer Science Department

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for Privacy

Damien D. Macielinski, Author

ACKNOWLEDGEMENTS

I would like to thank my committee members who have influenced my intellectual growth and personal development throughout my studies and work at Oregon State University. Thank you, Walter Rudd; your continuous guidance and influence as my major professor was indispensable. Thank you, Cherri Pancake, for expanding my horizons and including me in your group efforts. Thank you, Mike Quinn, for your lessons and influence in parallel computing.

Special thanks to Santhosh Kumaran for his brilliant input to my project and examples.

Special thanks to James Reinders for encouraging me to pursue a Master's degree. Where would I be without it?

Thanks to my parents, Ostap and Anna, for backing all of my decisions. Thanks for starting me on the right foot and making sure I stayed there.

TABLE OF CONTENTS

	<u>Page</u>
1 INTRODUCTION	1
1.1 Data-Parallel Programming	2
1.2 Task-Parallel Programming	2
1.3 Data-Parallel and Task-Parallel Programming.....	3
1.4 Types of Parallel Applications	3
1.5 The Need for Paradigm Integration	4
1.6 Objectives	5
2 PREVIOUS WORK.....	7
2.1 Level 1: A Message Passing Environment on MIMD Computers	8
2.2 Level 2: Adapt and Assign: Application-Oriented Parallel Compilers	9
2.3 Level 2: Dataparallel C with Modules.....	9
2.4 Level 2: Fortran M with HPF.....	10
2.5 Level 3: FX (Fortran-X).....	11
2.6 Level 4: ANSI Parallel Extensions for Programming Language C.....	12
2.7 Summary	13
3 DATAPARALLEL C OVERVIEW.....	15
3.1 Dataparallel C: The Programmer's View.....	15
3.2 Dataparallel C: The Compiler's View.....	16
3.3 Pseudo Task Parallelism in Dataparallel C: The <code>if</code> Statement.....	17
3.4 Resource Allocation.....	22
3.5 Summary	22
4 LANGUAGE EXTENSIONS.....	23
4.1 New Syntax.....	23
4.2 Data Dependency	25

TABLE OF CONTENTS (Continued)

	<u>Page</u>
4.3 Scheduling Tasks	28
4.4 Compiler Feedback	30
4.4.1 Schedule Output.....	30
4.4.2 Dependency Flowgraph Output	31
4.5 Summary	32
5 LANGUAGE IMPLEMENTATION.....	33
5.1 Communication Libraries	33
5.1.1 Eliminating the First Assumption	33
5.1.2 The Second Assumption	34
5.2 Statement Scanning and Parsing	35
5.2.1 New Tokens	35
5.2.2 New Productions.....	35
5.3 Example Source Code.....	36
5.4 Collecting Information.....	37
5.5 Creating a Flowgraph.....	38
5.6 DFS and Topological Sort.....	38
5.7 Scheduling.....	39
5.8 Code Generation	42
5.8.1 Static Tables.....	42
5.8.2 Assigning Processors	43
5.8.3 Communication Optimization.....	45
6 USING THE LANGUAGE: A SIMPLE EXAMPLE	47
6.1 Double Matrix Multiply Using Only Data-Parallelism	47
6.2 Adding Task Parallelism	48
6.3 Timing Results	51

TABLE OF CONTENTS (Continued)

	<u>Page</u>
7 BANDED SYSTEM SOLVER.....	53
7.1 Introduction.....	53
7.2 Parallelization	54
7.3 Solving A Banded System of Linear Equations	55
7.4 Implementation of the Solution Technique.....	60
7.4.1 Data-Parallel Solution.....	61
7.4.2 Augmenting the Example with Task-Parallelism	61
7.5 Summary of Findings.....	61
7.5.1 Purely Data-Parallel Implementation.....	62
7.5.2 Combined Task-Parallel and Data-Parallel Implementation.....	64
7.6 Chapter Summary	66
8 CONCLUSION.....	67
8.1 Conclusion	67
8.2 Future Work	68
8.2.1 Dynamic Virtual Processors.....	68
8.2.2 Scheduling Heuristics	69
8.2.3 Automatic Data Dependency Analysis	69
8.2.4 Nesting	69
8.2.5 Run-time Feedback.....	70
8.2.6 Automatic Task Compilation	70
BIBLIOGRAPHY.....	71
APPENDICES	74
APPENDIX A Source Code for Double Matrix Multiply.....	75
APPENDIX B Source Code for the Banded System Solver	77

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
2-1 Example pfor usage in ANSI X3H5 C.....	12
2-2 Example psections usage in ANSI X3H5 C.....	13
3-1 A Dataparallel C if statement: updating a poly variable	18
3-2 A Dataparallel C if statement: updating the same mono variable	19
3-3 A Dataparallel C if statement: updating different mono variables.....	20
3-4 An if statement without the as-if-serial rule applied	21
4-1 Example psections statement usage.....	24
4-2 Example IN and OUT statement usage.....	26
4-3 Ambiguity in dependency lists.....	27
4-4 Example ON statement usage.....	29
4-5 Compiler feedback - scheduling	31
4-6 Example dependency graph visualization with xvcg.....	32
5-1 Language extension productions.....	36
5-2 Pseudocode for examples in this chapter	37
5-3 Scheduling illustration	40
5-4 Example static table information	43
5-5 Simplified example of generated code.....	44

LIST OF FIGURES (Continued)

<u>Figure</u>	<u>Page</u>
6-1 Data-parallel matrix multiplication.....	48
6-2 Task-parallel matrix multiplication.....	50
6-3 Timing results for double matrix multiply (1120 x 1120 doubles).....	52
7-1 System of linear equations.....	56
7-2 Rearranged input matrix	57
7-3 Sub-Matrices within the rearranged input matrix	58
7-4 Pseudocode for repeated matrix-vector multiplications	60
7-5 Speed vs. Processors for all implementations.....	63

PREFACE

Organization of this Document

- Chapter 1 provides a brief introduction to data and task parallelism and the integration of the two.
- Chapter 2 provides related work by summarizing previous languages and methods of integrating task and data parallelism.
- Chapter 3 gives a brief introduction to Dataparallel C and discusses how task parallelism is related to the language.
- Chapter 4 summarizes the language extensions to Dataparallel C. This chapter functions as an informal user manual.
- Chapter 5 delves into the intricacies of implementing the language extensions.
- Chapter 6 provides a simple example which shows one way to use the language in order to improve the performance of a parallel program.
- Chapter 7 reports my finding in coding a real parallel application using the language extensions.
- Chapter 8 provides a conclusion and ideas for further research in this area.
- Appendix A provides source code for the example in Chapter 6.
- Appendix B provides source code for the example in Chapter 7.

Terminology

A *paradigm* is considered to be a set of methods found to be effective in handling a certain types of problems [1]. Two parallel programming paradigms are:

- **Data-parallel** - applying the same operation simultaneously over a data set.
- **Task-parallel** - applying possibly different operations over one or more data sets.

More detailed definitions and explanations can be found in Chapter 1. Data-parallel programming is synonymous with “domain decomposition”. Task-parallel programming should be considered synonymous with “control-parallel”, “control decomposition”, or “functional decomposition”.

Two of the most important architectures used in parallel computing are:

- **SIMD** - Single Instruction Multiple Data. Every processing element executes the same instruction in lock-step fashion (that is, instruction execution is fully synchronized).
- **MIMD** - Multiple Instruction Multiple Data. Processing elements execute independently and synchronization must be coded explicitly in the program.

There are two types of MIMD computer, based on how the memory layout appears to the user:

- **Multiprocessor** - The total memory of the system appears to be centrally located. All addresses are equally accessible from every processor.
- **Multicomputer** - The memory of the system is physically distributed among the processors. Communication and synchronization are achieved through message passing (at some level).

Conventions

- Pseudocode or actual code is in courier font.
- Variables in *equations* are italicized.
- **MATRIX** names are capitalized; vector names are lower-case.
- **PARAMETER** names are capitalized and boldface.
- $O(p(n))$ describes an algorithm whose complexity is on the order of some polynomial in n .

Task-Parallel Extension of a Data-Parallel Language

CHAPTER 1 INTRODUCTION

The introduction of high performance computing has led to a multitude of proposed and implemented parallel architectures. As evidenced by Pancake in [16], the high performance computing industry is still expanding. The technology and availability of parallel systems has increased dramatically, while the software accompanying parallel systems has not demonstrated a similar “ramp-up” in development [14]. This disparity has led to a concerted effort from the research community to provide some form of all-encompassing software to enable efficient use of the latest systems.

This thesis deals with the software which connects the user’s problem to the hardware: the programming language. High-level programming languages have been prevalent since the dawn of computers, when programmers quickly realized how difficult it was to code in assembly language. When used for high-performance computing, standard programming languages fall short of meeting the needs and expectations of the parallel programmer. Once again, users are finding it too difficult to attain optimal performance. Difficulties include:

- identifying parallel portions of an algorithm
- finding the optimal data distribution (granularity)
- finding the optimal task concurrency (load balancing)
- determining the minimum number of processors for optimal performance
- knowing details about the underlying hardware
- receiving performance feedback from an executing program
- coping with non-deterministic execution

The frenzy with which the research community has been developing new parallel languages (some based on old languages) has led to a myriad of high-level languages for

high performance computing [15]. Each has its own unique contribution for the parallel programming community. The language we have implemented combines two forms of parallel programming simultaneously: data-parallel and task-parallel programming.

The remainder of this chapter will discuss data-parallel and task-parallel programming, and the integration of these paradigms. We will also present a taxonomy used for classifying parallel applications. Finally, the objectives of our work are discussed.

1.1 Data-Parallel Programming

Data-parallel programming is defined as “the simultaneous application of a single operation to a data set” ([9] p.1). This means that each processing element is simultaneously executing an identical task on different portions of a data set. Information is exchanged at well defined synchronization points where every processing element must meet. Many problems in the real world are inherently data-parallel in nature.

1.2 Task-Parallel Programming

Task-parallel programming is the simultaneous application of (possibly) different functions to a data set (or several data sets) simultaneously. It lies in “the ability to translate a program into a set of functions to be applied in parallel” ([18] p.3). Information is also exchanged at synchronization points, but only the sender and receiver(s) need to observe a particular synchronization point. This form of parallel programming is difficult to understand and has problems with scalability and determinism. Nevertheless, some applications require task parallelism for efficient execution. Signal processing, multidisciplinary, and asynchronous applications are well suited for task parallelism.

1.3 Data-Parallel and Task-Parallel Programming

There is some ambiguity surrounding the relationship between task and data parallelism. One definition of task parallelism states: “Each processor is doing a completely different thing” ([5]). We disagree, since a subset of processors performing the same function may be executing concurrently with other processors performing a different function. This case is still task parallelism even though some processors are doing the same thing. Note that a task parallel application which has all processors executing the same function degenerates to data parallelism when all processing elements participate in all synchronization points and there is a synchronization point after every statement or block of statements. Therefore, data parallelism is a special case of task parallelism.

1.4 Types of Parallel Applications

Geoffrey Fox ([7]) proposed a classification system for parallel applications. The system takes into account the programming paradigm and the type of architecture for solving the problem. His system has three main classes:

- **Class I - Synchronous Applications.** These programs are temporally and spatially regular. Synchronization is performed at the individual instruction level. A SIMD architecture is a natural target for these applications. These applications fit well into the data-parallel programming style.
- **Class II - Loosely Synchronous Applications.** These programs are similar to those in Class I, except that the synchronization is performed only when it is needed to coordinate the activities of the processing elements. Hence, these applications are temporally irregular and spatially regular. The natural target architectures for these systems is MIMD with distributed memory. Hatcher & Quinn [9] have illustrated how these programs also map onto a data-parallel style. The programmer maintains a SIMD view of the program, while the compiler (such as C* [24] or Dataparallel C [9])

is able to loosen the synchronization and bring the compiler output program into Class II. This allows a migration path for Class I programs onto MIMD architectures. This has allowed significant performance improvement for many data-parallel applications.

- **Class III - Asynchronous Applications.** These applications are spatially and temporally irregular. The natural target architecture for these applications is unclear (Fox suggests the possibility of MIMD or Shared Memory). The programming style for these applications must be task-parallel. Data-parallel schemes require a degree of synchronization which would not allow this class of applications to execute at maximum speed.

Fox also claimed that "... many complicated problems are mixtures of the basic classification". This led him to propose a classification that lies between II and III called **IIICG-IIIFG**. This is a Course Grained asynchronous program controlling Fine Grained loosely synchronous subproblems. In other words, it is a task-parallel framework controlling data-parallel functions (because the tasks being executed concurrently are themselves data-parallel routines).

A language that efficiently compiles programs from the "mixed" class may prove to be the bridge between Class II and Class III (much as C* and Dataparallel C have bridged Class I and Class II). This language needs to integrate both data-parallel and task-parallel programming styles with the ability to nest data parallelism inside task parallelism. This language allows some Class II applications to join Class IIICG-IIIFG with significant performance improvement (like the applications that moved to Class II from Class I via Dataparallel C or C*).

1.5 The Need for Paradigm Integration

A parallel language which encompasses both data-parallel and task-parallel programming paradigms is necessary for efficient and simple coding of type IIICG-IIIFG programs. Current data-parallel languages are too restrictive and do not allow different functions to be executed concurrently with minimal synchronization. A different style is needed to ease these restrictions and allow an integrated style of parallel programming.

Having both paradigms in the same language will allow programmers to choose a purely data-parallel or task-parallel style, as well as the ability to alternate between the paradigms and nest the paradigms at will. Having these features in one language has several advantages:

- **Ease of use:** Only one language needs to be written and debugged. No communication or synchronization primitives need to be explicitly coded (the compiler generates these).
- **Flexibility:** The programmer chooses the programming style, and is given the ability to nest styles.
- **Power:** Some algorithms do not execute efficiently in a purely data-parallel or purely task-parallel environment. The combination of paradigms will give the programmer the opportunity to efficiently execute their programs.
- **Portability:** By eliminating low level system routines (for message passing or synchronization) the compiler will be able to target almost any system.
- **Code Reuse:** By using one language, libraries of common parallel subroutines may be written.
- **Optimization:** Using one compiler provides the ability to balance the important trade-offs in using both paradigms simultaneously ([22],[23]).

1.6 Objectives

Given the advantages of data-parallel programming and the prevalence of data-parallel programming languages, we decided to add the necessary features to exploit task parallelism in an existing data-parallel language. By adding functionality to an existing language, we keep the syntax, semantics, and functionality of the original language. Programs previously written in the language still compile and execute correctly, and become amenable to simple modifications to exploit any task parallelism. Also, any architectures targeted by the original language are also targeted with our modifications.

We used Dataparallel C as the base language for the extensions because of our familiarity with this language and previous exposure to the compiler. The compiler

sources are readily available, and we have contact with the current maintainers of the language.

We will also show how this language provides a migration path for applications from Class II programs into Class III CG-IFG. The language improves the performance of some data-parallel programs by allowing programmers to exploit concurrency among data-parallel operations. We have strived to make this migration easier by providing simple task-parallel extensions to Dataparallel C. All programming is done within one language. The user does not need to be concerned with explicit message passing, modules, channels, or processor allocation. Since the compiler handles these low-level details, the programmer need only be concerned with efficient implementation of parallel algorithms.

CHAPTER 2

PREVIOUS WORK

In this chapter we describe some of the previous attempts at combining the two parallel programming paradigms discussed in Chapter 1. We present this overview in a manner that describes the evolution of these methods. The evolution of this class of parallel languages has been at four conceptual levels:

1. Message passing libraries
2. Communicating modules
3. Compiler directives
4. Language extensions

The first level encapsulates the oldest method for writing parallel programs on a MIMD computer. It uses a sequential language (C or Fortran) with libraries that facilitate the injection and reception of data onto/from the computer's network. This is achieved through send and receive function calls (at the lowest level).

The second level, modules, abstracts from the first by allowing the programmer to think in terms of parallel operations and their necessary inputs and outputs. The user solves smaller problems with groups of processors (sometimes one) and connects them with channels. Channels are simplifications built on top of the message passing method.

The third level, directives, tries to abstract away any notion of connections. The user typically uses a traditional language with the addition of "hints" to the compiler. These hints are in the form of compiler directives that specify data distribution (for data parallelism) or the necessary values for input before a computation can begin (for task parallelism).

The fourth level builds on the third by adding language constructs to a base language (e.g. special looping syntax and semantics). Directives may still be used. This level is perhaps the closest to a truly parallel language because parallel expression is part of the language semantics.

The remainder of this chapter will cite examples of languages from each level. All the languages are sufficiently powerful enough to express both task and data parallelism.

2.1 Level 1: A Message Passing Environment on MIMD Computers

This is the most well known method of parallel programming. Every distributed-memory MIMD supercomputer provides the user with some way of programming at this low level. Users are required to write a separate program for each processor they will be using. The programs include calls to library routines that can send (receive) messages to (from) the supercomputer's interconnection network. Messages are used for two purposes: to transmit data and to synchronize activities between processors.

Data parallelism is achieved by writing an identical program for all of the processors. In a typical application, data is distributed to each copy of the program, and computation begins. Eventually, the processors need to communicate their data, so each issues a send, with a matching receive command at the other end. Once the data has been received, computation continues. Eventually, a final result is collected, and all the programs halt.

Task parallelism requires separate programs or control flow for each of the tasks to be performed. Each task issues a receive command and waits for the data to arrive. Once all data is available, the task may begin execution. When completed, the task sends its results on to the next destination.

Programming with explicit message passing is extremely tedious. Coding a data-parallel program requires just one program for all the processors, while task parallelism potentially requires separate programs. The message passing calls need to know processor numbers, size of the message, and the actual data in the message. Some systems even require the specification of a route through the network. Not only is the programming tedious, but code development takes a long time, and scalability is virtually nonexistent. The hardest part is debugging an application, especially when the problem involves deadlock resolution or a non-deterministic sequence of events.

2.2 Level 2: Adapt and Assign: Application-Oriented Parallel Compilers

Intel has developed two programming tools that are special purpose or “niche” compilers ([18]). **Adapt** is used for image processing applications, and is a data-parallel language. The user specifies routines that are to be applied simultaneously to an input image. **Assign** is used for signal processing applications, and is a task-parallel language. The user specifies functions that are to be applied to streams of data. The functions are connected by paths representing data dependencies between the functions.

Both of these languages target an intermediate Level 1 style interface called PCS (Programmed Communication Service). This layer specifies the message passing between processors, and hierarchically groups operations into modules. These modules may later be connected with another PCS program.

To combine the two paradigms, programmers must compile task and data parallel modules separately with the appropriate compiler. They must then write a PCS program to instantiate and connect the modules with channels that will carry the data among the modules.

The entire procedure is time consuming and convoluted (perhaps because no tool is provided that can simplify this procedure). The user must be familiar with three different programming environments in order to achieve an integration of task and data parallelism. Scalability and debugging are problematical, since the modules are not aware of the existence of other modules.

2.3 Level 2: Dataparallel C with Modules

Seevers, Quinn, and Hatcher [19] devised a method to combine Dataparallel C programs via channels to allow data parallelism to be nested in a task-parallel control structure. This environment requires the programmer to explicitly code any data transmission between modules with `fread` and `fwrite` commands to an opened channel. Therefore, modules must agree on their communication patterns at compile time.

The programmer must also write a “link file” using a control language, and pass this information onto a module loader and channel linker. This tool determines processor allocation and channel construction, all at compile time.

Once again, programming is tedious because the user must write a separate program for each module, then write a link file. Although not as tedious as Level 1 type programming, the user must still open and close channels which are read from and written to explicitly. This practice is error prone and hinders the scalability of programs.

The language we are proposing also extends Dataparallel C, but in an entirely different manner. The user writes only one program and never has to worry about message passing or channels. The user specifies only data dependencies, which the compiler converts into explicit message passing. The programs are completely scalable with minimal effort on the programmer’s part.

2.4 Level 2: Fortran M with HPF

Fortran M ([4],[5]) is a task-parallel language which uses modules to connect operations via channels. The advantage over the Assign/Adapt/PCS environment and the modular Dataparallel C environment is that both programming and module connection are done in a single language (Fortran with language extensions).

HPF (High Performance Fortran) is a set of extensions to Fortran 90 which support data-parallel programming. Special directives are used to allocate processing elements, align data, and distribute data. A `forall` statement is then used to execute operations in parallel.

The integration of task and data parallelism lies in the combination of these two languages ([2],[6]). More specifically, HPF data-parallel subroutines are wrapped with a subroutine that controls data input/output via Fortran M channels. A Fortran M program is then used to invoke the HPF routines from within a `processes / endprocesses` section which allows concurrency among the calls in the section. Hence, a task parallel

framework (written in Fortran M) is used to coordinate data-parallel modules (written in HPF).

This approach is tedious because two languages must be used. Nevertheless, the combination is relatively new, and the developers have plans to incorporate the two in a more coherent structure.

This language may be considered Level 1, because the user must explicitly send and receive data through ports. The advantages over Level 1 is that these commands are not machine dependent, and that the compiler may perform type checking between the ports. This language might also be considered Level 3 because of the directives used in HPF. It might also be considered Level 4, because of the language extensions provided by Fortran M and HPF. Nevertheless, the main thrust of the language is to allow programmers to define individual modules which may be arbitrarily combined or reused. The methods used to achieve this modular interconnection have implications from Levels 1, 3, and 4, but we will consider this a Level 2 language.

2.5 Level 3: FX (Fortran-X)

FX ([20],[21]) is another Fortran based parallel language, developed at Carnegie Mellon University. Task parallelism is attained through the use of compiler directives that specify input and output values for each task. Directives are also used to identify the sections of the code that contain parallel constructs or subroutines. Data parallelism “is in the form of independent parallel loop iterations” ([20] p. 6). Each task subroutine may be data-parallel, and hence data parallelism may be nested inside task parallelism.

The programmer is not concerned with ports, channels, messages, or modules; these are handled transparently by the compiler. Therefore, the language resides purely in Level 3. The ability of the compiler to handle low-level programming tasks is a much needed benefit for the simplification of parallel programming. It will also be easier to support debugging and portability between systems.

An interesting feature of this compiler is the ability to generate efficient schedules for programs that have data parallelism nested in a task-parallel framework ([22],[23]). The code is executed at most 5 times in purely data-parallel and task-parallel forms. Run-time parameters are collected and used to determine an optimal final mapping onto the system. FX also has the ability to replicate parallel operations that do not scale well with the addition of processors. The replicated pieces take turns computing results for each iteration, thereby increasing throughput.

2.6 Level 4: ANSI Parallel Extensions for Programming Language C

An attempt is currently being made towards a standardization of parallel languages. One group is concerned with the addition of language constructs to the ANSI C language ([25]). The proposed language is at a very high level; the programmer need not be concerned with channels, modules, or special directives. Variables may be declared as shared (among processing elements) or private (local to each processing element). All parallelism is expressed through natural language constructs. For example, a `for` loop which has the *potential* for executing the loop body concurrently is shown in Figure 2-1. The executions of `f1` may be done in a data-parallel style if the iterations are independent of one another.

```
parallel {
  pfor(i=0; i < ITERS; i++; latch) {
    f1();
  }
}
```

Figure 2-1. Example `pfor` usage in ANSI X3H5 C

Task parallelism is expressed through a `parallel` and `psections` statement. Figure 2-2 shows how three (possibly different) functions may be executed concurrently. Data and task parallelism may be arbitrarily nested, and subroutines may themselves contain parallel sections.

```
parallel {  
    psections {  
        f1();  
        f2();  
        f3();  
    }  
}
```

Figure 2-2. Example `psections` usage in ANSI X3H5 C

Despite its apparent simplicity, this language requires the user to handle process synchronization explicitly. If synchronization is overlooked, or mistakes are made, the program will not execute correctly.

Several supercomputing companies (including KSR and Convex) have implemented languages similar to, or based on, this standard. It appears that this language is most conducive to systems that support efficient dynamic process creation and migration (as opposed to compile-time allocation of resources).

2.7 Summary

The languages presented above illustrate the variety of methods used to express data and task parallelism within a single framework. We have attempted to delineate these languages based on what the programmer needs to do to express parallelism. We have

divided them into four levels. The most difficult to program is Level 1, with the easiest being in Level 4. The trade-off is that the user has more control at the lower levels at the cost of more tedious programming.

Several implementation styles can be derived from the languages above. There are trade-offs associated with each:

- Run-time vs. compile-time allocation of processing elements. Some languages incorporate both, or gather run time information for compile time allocation.
- Fortran vs. C language base. Fortran has been preserved for the sake of scientific programmers and “dusty-deck” codes. C has also been used for those who have moved to a newer programming language.
- Directives vs. Constructs. Directives allow any original language syntax to be used without modification. Language constructs allow new programs to be written in an easier and more consistent manner.
- Synchronization vs. Data Dependence. These are two different methods which allow the user to express the correct order of operations.

The language we have implemented is discussed in more detail in the forthcoming chapters. The trade-offs discussed above are resolved in the following manners:

- Task and data parallelism are incorporated by extending an already existing data-parallel language (Dataparallel C) with task-parallel constructs. The constructs are modelled after the ANSI proposal discussed above.
- C is the base language used (instead of Fortran) because of our previous experience with C compilers, and the availability of Dataparallel C for our work.
- Scheduling and processor allocation is resolved at compile-time. The systems available to us did not have efficient mechanisms for the creation, reuse, and destruction of parallel threads of execution at run-time. Dataparallel C already uses compile-time allocation, and it is worthwhile to remain consistent with the base language’s methods.

The next chapter describes Dataparallel C from the user’s and the compiler’s perspectives. A discussion of the Dataparallel C `if` statement is included to show how a programmer may try to induce task parallelism in Dataparallel C.

CHAPTER 3

DATAPARALLEL C OVERVIEW

This chapter provides a short introduction to Dataparallel C. First the programmer's role is summarized, followed by a short explanation of the inner workings of the compiler. Finally, an investigation of the Dataparallel C `if` statement is presented. The `if` statement is important in understanding how a programmer may try to induce task parallelism in Dataparallel C, and how the language will not allow true concurrency. Without true concurrency, something else must be done to allow task parallelism in Dataparallel C. Chapter 4 will then discuss the language extensions that allow true concurrency.

3.1 Dataparallel C: The Programmer's View

The brief description included here is by no means complete. The intent is to define some terms and syntax that are used in the remainder of this thesis. A more thorough explanation of Dataparallel C may be found in [9].

The Dataparallel C programmer keeps in mind a vision of a SIMD style architecture while programming. This implies that there is a front-end uniprocessor connected to numerous "smaller" back-end processors. The sequential portions of the program appear to execute on the front end, while the parallel portions appear to execute on all of the back-end processors. The programmer selects the number of active processors on the back-end, and these are called *virtual processors*.

Virtual processors are used as abstractions of the real processors in the system. Therefore, the programmer may select any arbitrary number of virtual processors without having to consider the actual number of physical processors on the underlying system. Each virtual processor has an identical memory layout. Variables which are local to each virtual processor (called *poly data*) are declared in a *domain declaration* section. Variables

not declared in a domain declaration are *shared variables* (also called *mono* data), and appear to reside in the front-end memory.

Poly variables have scope only inside a *domain select* statement. The front-end waits while the back-end processors synchronously execute the code inside the domain select statement. Both poly and mono data may be accessed here, except that writing to a mono or accessing another virtual processor's domain data will incur a communication penalty, slowing the resulting computation.

Functions declared inside a domain declaration are called *member functions*. When called from parallel or sequential code, the member function is executed by all active virtual processors. The member functions have access to the variables declared in the corresponding domain declaration.

Sequential (non-member functions) may also be called from sequential code or from parallel code. When invoked from sequential code, the body of the function is executed by the front-end processor. If called from parallel code (inside a domain select), each virtual processor executes the function body.

3.2 Dataparallel C: The Compiler's View

Despite the SIMD view presented to the programmer, the actual target architecture is a MIMD multicomputer, a multiprocessor, or a network of heterogeneous processors. Virtual processors are assigned in a many-to-one fashion to the physical processors of the underlying system. Every physical processor executes the sequential code, and parallel code is modeled by a virtual processor emulation loop. This is a `for` loop which iterates over all of the virtual processors assigned to a physical processors. Each iteration executes the necessary code for each virtual processor. Hence, lock-step synchronous execution is only emulated. Synchronization (communication) points are introduced at appropriate points in the program, where active virtual processors can update non-local variables.

Poly variables (from the domain declaration) are represented as an array of variables with one entry for each virtual processor on the physical processor. Shared variables are

maintained as a single copy on each physical processor. Whenever a mono variable is read, the local copy is used. Whenever a mono variable is written, all processors must communicate in order to update their local copies.

The next section describes the semantics of the `if` statement and how it relates to task parallelism.

3.3 Pseudo Task Parallelism in Dataparallel C: The `if` Statement

One place of particular interest is the `if` statement. A programmer may try to force different virtual processors into different pieces of code by branching on the virtual processor number. This may appear to induce task parallelism, but the `if` statement must abide by the *as-if-serial* rule. This means that any virtual processors that evaluate the conditional expression to true execute the *then* clause of the statement. If a shared variable is updated in the *then* clause, then all other virtual processors must wait until this portion is completed. Once the *then* clause is finished, a communication occurs (1) to make sure all processors are synchronized and (2) to communicate the new values of any shared variables which were modified by the *then* clause. Next, the virtual processors that evaluate the condition to false execute the *else* clause while the others wait (assuming a shared variable is updated in the *else* clause). Then another communication is performed at the end of the *else* clause. Notice that this allows arbitrary nesting of `if` statements inside each other. Because of the *as-if-serial* rule, true task concurrency is not possible because all of the physical processors participate in each of the statements in the *if-then-else* block.

Figure 3-1 shows a portion of a Dataparallel C program and its corresponding translation (by the compiler) into the intermediate representation (C with message passing) that is executed on each physical processor. The translation has been greatly simplified for readability. The value being modified by the `if` statement is a poly, so every virtual processor maintains its own copy. This case does *not* apply to the *as-if-serial* rule, because each virtual processor may perform the update simultaneously without interfering

with the value maintained by other virtual processors. There is no synchronization between the clauses of the `if`, since the new values do not need to be broadcast to the other virtual processors.

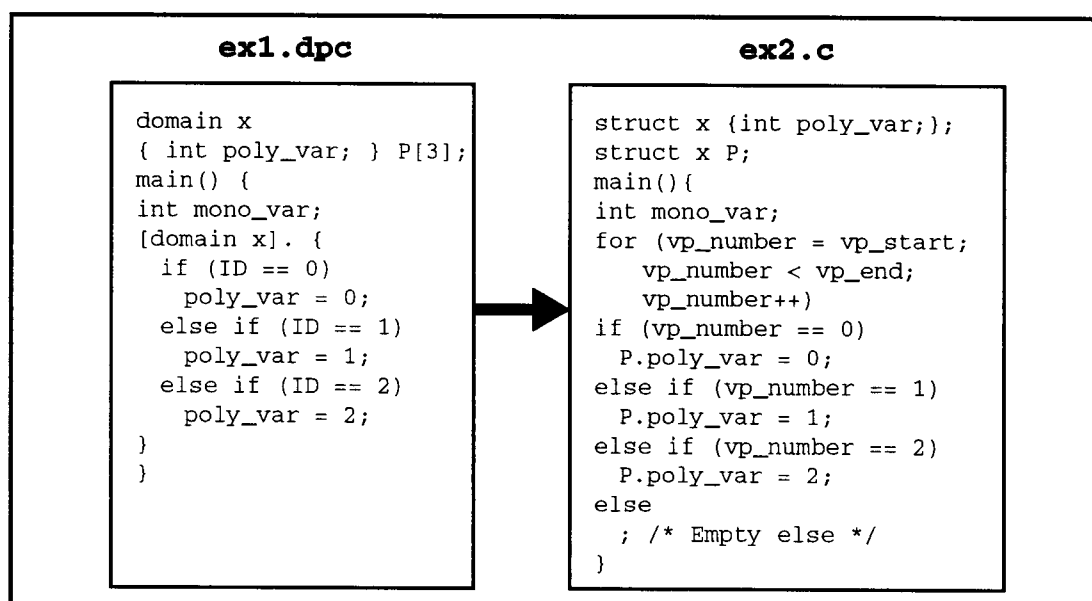


Figure 3-1. A Dataparallel C `if` statement: updating a poly variable

Figure 3-2 shows another case, where the value being updated is a mono variable (the same one in each clause). This is the particular case of interest, when each of the virtual processors may be updating a shared variable. Any virtual processors that determine the `if` statement to be true execute the *then* clause, while the virtual processors determining it to be false have to wait until all of the true virtual processors have completed their work. Then a synchronization will take place, and any shared variables are reduced to a single value and broadcast to all of the physical processors for an update. Now the first group must wait while the false virtual processors execute the `else` clause. Finally, another synchronization is done to reduce and broadcast any shared variables modified in the `else` clause.

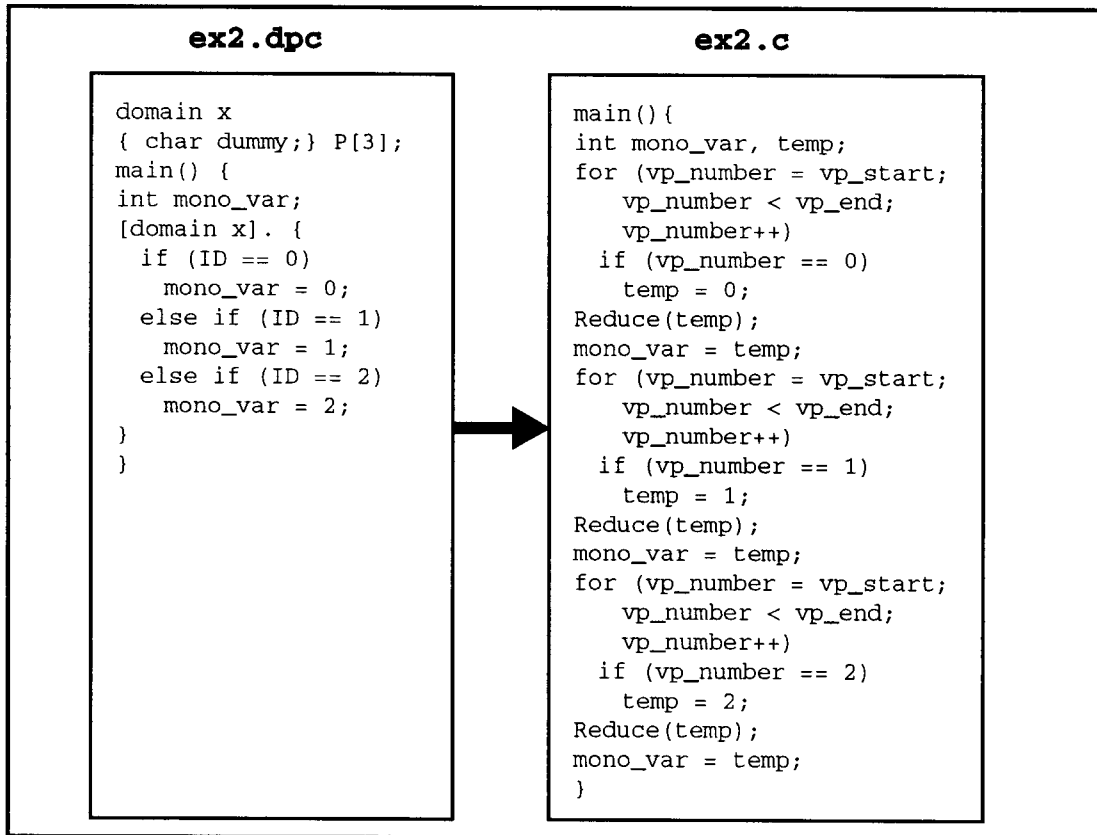


Figure 3-2. A Dataparallel C `if` statement: updating the same mono variable

The main difference between Figure 3-1 and Figure 3-2 is the communication introduced by updating a shared variable in the call to `Reduce`. This routine will collect all of the values of the shared variable being updated. One value is chosen and broadcast to all of the physical processors so that they will all have identical copies. The reduction is done after each clause in the three stage `if` statement in Figure 3-2. This communication is very time consuming, but is a necessary evil to guarantee correct results by the as-if-serial rule.

The next example is a slight variation on the last one. What if the shared variable being updated in each of the clauses is different? The code generated is shown in Figure 3-3 and is basically identical to that in Figure 3-2, except that the new mono variable names are used.

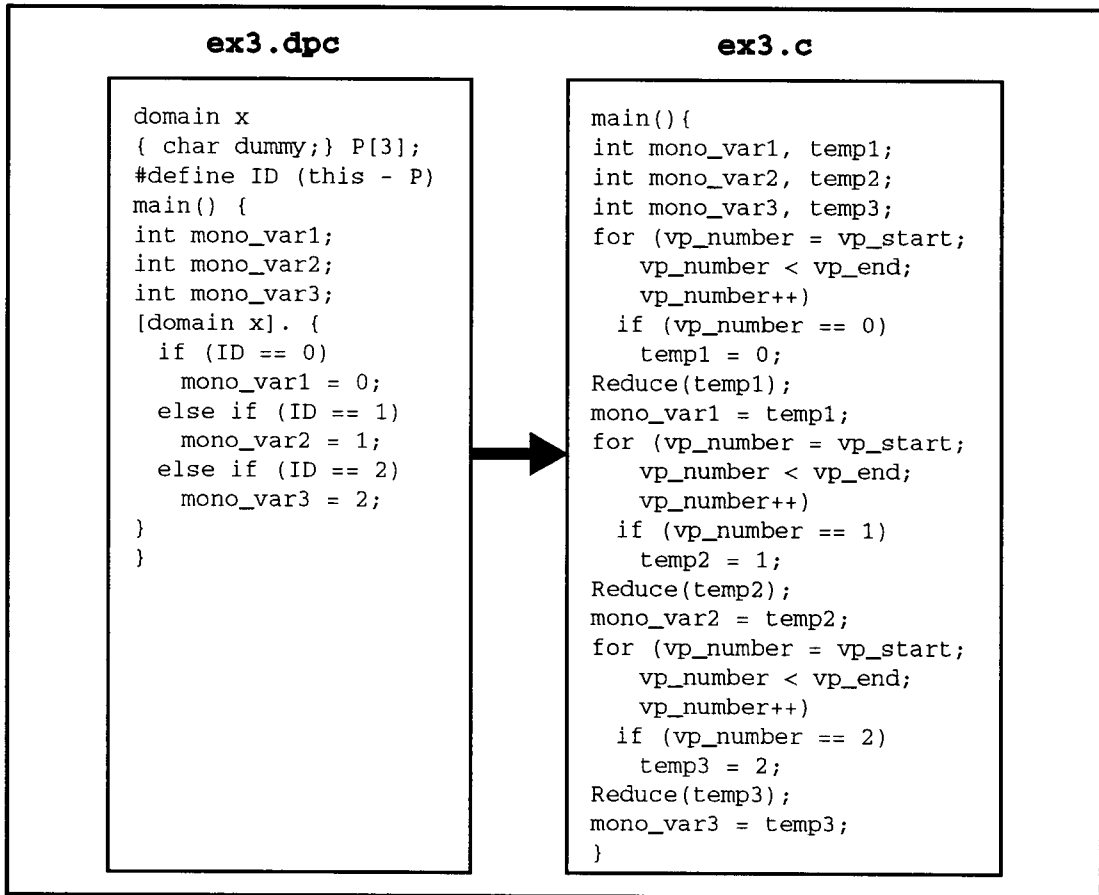


Figure 3-3. A Dataparallel C `if` statement: updating different mono variables

The code generated in this example is important. It shows how programmers would try to achieve task parallelism in a data-parallel language. They would assign work to virtual processors based on virtual processor numbers, and mask off the remaining processors. Each task may access shared variables in a manner that is non-intrusive to the other processors. Unfortunately, because of the as-if-serial rule, there is no concurrency among the `if` statement blocks, and true task parallelism cannot be achieved.

Since the three blocks update different shared variables, the communication (reduction) is not really necessary until *after* the entire `if` statement. Figure 3-4 shows what this code would look like, without the as-if-serial rule applied. Notice that the shared variable assignments (tasks) are now done concurrently, and the new values are broadcast

at the end of the statement (once as opposed to three times). Also notice that only one virtual processor emulation loop is needed for each physical processor.

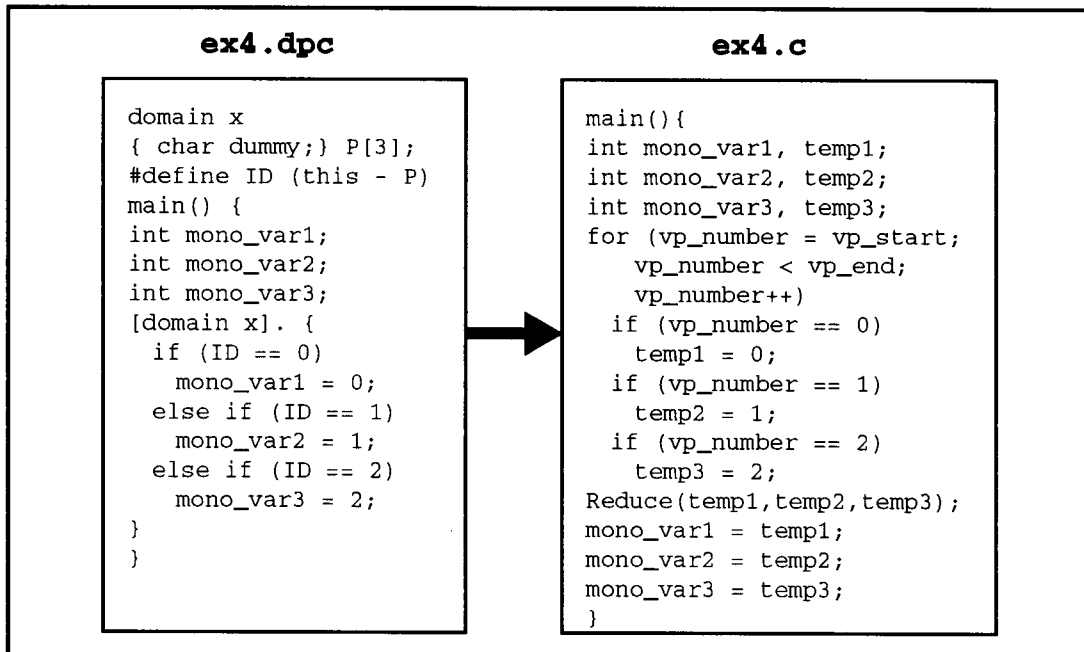


Figure 3-4. An `if` statement without the as-if-serial rule applied

“Loosening” of the as-if-serial rule is one method of allowing task parallelism in Dataparallel C. Unfortunately, this method of task-parallel programming (Figure 3-4) is tedious and inefficient. The programmer must be concerned with which operations are assigned to which virtual processors. The final mapping of virtual to physical processors is not controllable by the user. Therefore, the order of assignment of tasks to virtual processors has an impact on program performance.

Also note that in order to achieve complete concurrency, no communication can be allowed inside the `if` clauses. This is an unrealistic restriction since concurrently executing data-parallel operations must communicate when operating in parallel.

Therefore, real task-parallel programming cannot be “induced” in Dataparallel C in its current state.

3.4 Resource Allocation

An interesting side effect results from the as-if-serial rule. We call this the *all-or-one* rule. The processors allocated to the program execution execute in a well defined manner:

- In parallel code, *all* physical processors must cooperate in communication, even if they have no work to do. This follows directly from the as-if-serial rule.
- In sequential code, each processor computes exactly the same result as if only *one* processor were doing the work.

This method of allocating resources to tasks is too restrictive for task parallelism. After all, we would most likely want some intermediate number of processors (neither one nor all) to execute each of the concurrent tasks. Without loosening the restriction imposed by the all-or-one rule, true task concurrency cannot be achieved.

3.5 Summary

Our proposed language extension “loosens” the as-if-serial rule, but in a way that allows communication inside the concurrently executing sections. It also provides a more convenient syntax than the `if` statement. The all-or-one rule is also “loosened” since the number of resources dedicated to each task is allowed to vary anywhere from 1 to all of the processors. Assignment of resources to tasks is handled by the compiler after the user specifies how many resources should be allocated to each task (i.e. the user need not be concerned with explicit assignment of tasks to processors). The details regarding the syntax and language usage is described in the next chapter.

CHAPTER 4

LANGUAGE EXTENSIONS

In extending Dataparallel C, several issues must be resolved by the new language. The resolution of these issues influence the way in which the language is extended. Some of the issues (and their resolution) discussed below are:

- Expressing Concurrency
- Data Dependency
- Scheduling Tasks

Chapter 5 gives implementation details for these extensions. Chapter 6 provides an example program to show how the new language extensions can be used to add task parallelism to a data-parallel program.

4.1 New Syntax

Recall from the last chapter that we can express task parallelism in Dataparallel C with a *special* `if` statement which does not use the as-if-serial rule (while maintaining the rule for all other `if` statements). Two possible implementations have been considered:

1. A `psections` statement similar to the one proposed by the ANSI X3H5 committee in [25] (see “Level 4: ANSI Parallel Extensions for Programming Language C” on page 12).
2. A `pif` statement (parallel `if`) that is used exactly like the standard `if` statement, except that the as-if-serial rule does not apply.

Both statements are semantically equivalent, but the first one is easier to use. The programmer does not need to be concerned with explicit task allocation, which is done by the compiler. Also, a `pif` statement may imply a run-time partitioning of tasks to processors. This cannot happen since all scheduling is done statically at compile time. Therefore a `psections` statement is used to express the task parallel operations in the language extension.

The `psections` statement may appear anywhere in sequential code. The general usage is given in Figure 4-1. The open and close braces enclosing the `psections` block are mandatory. Each statement inside the `psections` can be a block, function call, or any legal compound statement. `psections` may not be nested, and may not appear in a domain statement.

```
psections {  
    statement1;  
    statement2();  
    { /* statement3 */  
        st;  
        st;  
        ...  
        st;  
    }  
    statement4;  
    ...  
}
```

Figure 4-1. Example `psections` statement usage

Each top-level statement inside the `psections` becomes a candidate for concurrent execution with all of the other statements inside the `psections`. Therefore, the statements can potentially be executed in an undefined order, unless data dependencies are specified to impose a “correct” order of execution.

4.2 Data Dependency

As mentioned previously, some tasks may have to wait for the completion of other tasks. For example, execution of task A results in a vector, which will become the input to task B. Therefore, tasks B and A cannot execute concurrently because task B depends on data from A.

The data dependency must either be expressed by the user, or detected by the compiler. A task graph results from this analysis, with nodes representing tasks and edges representing data dependencies. This graph is then used to schedule the tasks in a way that maximally exploits concurrency between tasks, and minimizes communication between processors.

Efficient data flow analysis for automatic dependency detection is needed when the user is not required to specify the dependencies. This is very difficult and time consuming to implement correctly, and is beyond the scope of this project. Nevertheless, investigating the addition of data-flow analysis for ease in programming is an interesting topic for future work (see “Future Work” on page 68).

Without data flow analysis, the user must provide the dependencies. Asking the user to specify data dependencies is not unusual. The Fortran M language ([4]) achieves this by making the user specify channels between tasks (much like the edges in the task graph). The CMU Fx compiler ([21]) uses compiler directives to specify input and output variables for each task.

Therefore, we do not consider user specification of dependencies to be unreasonable, especially for this project. Two possible implementation schemes exist:

- Compiler directives
- Dependency specification as part of the language construct

Compiler directives may be used since the dependence information is needed only at compile time (for scheduling), and no dynamic analysis need be performed. One advantage to directives is that they may be easily removed or ignored if the language evolves to support automatic compiler dependency analysis. The second alternative has the advantage of being more comfortable for programming. The dependency specification

is part of the language; hence the compiler can issue warning messages if the dependency is incorrectly specified.

The distinction between the two is minimal. We have chosen two language constructs that will allow the user to specify dependencies. Type checking is done by the compiler to make sure any variables specified in the dependency lists conform to language restrictions.

The programmer uses `IN` and `OUT` statements to specify a comma-delimited list of mono variables that are used in dependency analysis. Figure 4-2 shows an example of how these statements are used. Any values which are needed by a task in order to begin execution are listed in the `IN` statement. Any mono values which are computed as outputs to another task are listed in the `OUT` statement. If a task has dependencies, then the `IN` and `OUT` statements must appear before their corresponding statement inside the `psections` block. The `IN` and `OUT` statements are entirely optional since a task may not have input or output dependencies. There are no restrictions on the type of variable that can be used (as long as it is not a poly). Also note that the variables specified in the `OUT` statement are not available for use until the end of the statement.

```
psections {  
    IN x,y,z;  
    OUT a,b,c;  
    statement1;  
  
    IN a,b;  
    statement2;  
}
```

Figure 4-2. Example `IN` and `OUT` statement usage

Notice that there are no explicit ports or channels used to specify dependencies (like some of the languages cited in Chapter 2). The programmer need only specify the input and output variables to the corresponding statement. The major drawback of this system is the possibility of ambiguity. For example, Figure 4-3 shows an example where two statements have the same variable in their OUT lists (statements 1 and 3). Then a third statement has the variable in its IN list (statement 2). The compiler cannot determine which of the two OUT values is supposed to be sent to this statement. When ambiguity is detected, the compiler issues an error during compilation.

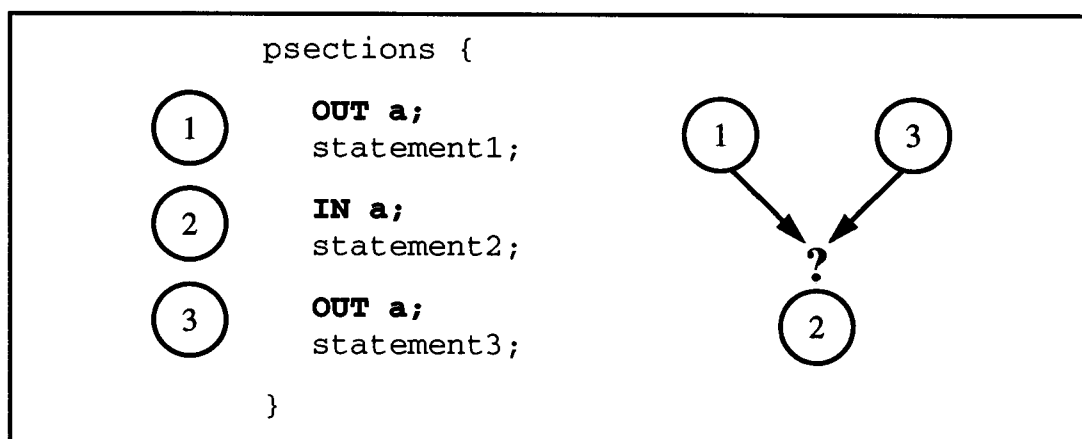


Figure 4-3. Ambiguity in dependency lists

A variable repeated in several IN lists does not constitute ambiguity (as long as it comes from only one OUT list). The compiler can detect this “fan-out” case and automatically send the data to all of the desired IN variables, wherever they may appear.

IN variables that do not have a corresponding OUT are called *dangling inputs*. The compiler assumes that these values are coming IN from outside the `psections` statement. Since these mono variables are already present on every processor, no special handling is needed. Dangling inputs are ignored.

Conversely, *dangling outputs* are OUT variables with no corresponding IN match. The compiler assumes that these values are to be updated to all of the processors at the END of the `psections` statement. Therefore, upon completion of the `psections`, all of the dangling OUT values are broadcast from one of their statement's processors to all other processors. The values being broadcast are not guaranteed to be updated until the end of the `psections` statement.

Using this scheme, it is also easy to create a cycle in the dependency list. Cycles are not allowed because the compiler cannot always determine the correct order of execution of the statements inside the `psections`. We have also stated that a task cannot begin execution until all of its inputs are available. In a cycle, this rule cannot be maintained. The compiler can detect cycles and will issue an error whenever one is detected (no matter how large or small).

4.3 Scheduling Tasks

Since the all-or-one rule for resource allocation in Dataparallel C (see "Resource Allocation" on page 22) has been eliminated, the compiler must know exactly how many processors need to be allocated to each task. Again, this is left to the programmer who must specify the number of processors via the ON statement. An example is shown in Figure 4-4. The ON statement must be used before each statement in the `psections` block. The number specified determines how many *actual* (not virtual) processors are to be devoted to the task. The number must be a constant expression and a power of two¹.

1. The power of two restriction is necessary because the Dataparallel C libraries are used for communication within a task. These communication primitives are implemented for use on a power of two number of processors.


```
#define P 16

psections {

    OUT a;
    ON 4;
    statement1;

    IN a;
    ON P/2;
    statement2;

}
```

Figure 4-4. Example ON statement usage

Allowing the user to specify virtual processors (instead of physical processors) might be the ideal way to allocate resources, but this causes a major problem in implementation when two virtual processors from the same physical processor are each assigned two different tasks to operate on concurrently. Synchronization, and hence message passing, should only occur within the context of each task. Therefore, the two virtual processors should compute independently of the other. This is not physically possible, since the two virtual processors are being emulated by one physical processor, which cannot execute two separate pieces of code simultaneously².

By forcing the user to specify *physical* processors in the ON statement, the problem presented above is resolved. Unfortunately, this solution has an undesirable side effect on the way a user must write programs which have data-parallel code nested inside a `psections`. The user must be aware of how many virtual processors were statically allocated to each physical processor. Then, when coding each task, the programmer must multiply the amount of work done (and space used) by a virtual processor so that all of the virtual processors in the task will emulate all of the virtual processors in the system. This

2. Multiple threads may be able to emulate this, but true concurrency would be limited by context switching. Also, Dataparallel C does not currently support virtual processor emulation with threads.

restriction does not limit scalability, but makes scalable code more tedious to write. This is evident in the toy program code given in Appendix A. “Future Work” on page 68 examines some possible methods for solving this problem.

4.4 Compiler Feedback

The compiler provides two forms of feedback to the user. It shows the generated schedule in a textual format and the dependency flowgraph in a graphical form.

4.4.1 Schedule Output

By allowing constant expressions in the ON statements, the programmer may adjust resource allocation on the command line when the compiler is invoked. This alleviates constant changing, saving, and compiling of the program to run on various sized systems or smaller numbers of processors within the system. Unfortunately, the user may lose sight of what the compiler is doing “under the hood” with resource allocation. Therefore, the compiler emits a simple Gantt chart depicting the schedule. The programmer can quickly scan the chart to ensure proper resource allocation for each compilation. An example of the compiler output is shown in Figure 4-5. Gaps in the schedule are filled in with “-1” values. All other values represent a statement numbering scheme which is also dumped with the schedule. Processors are numbered in Gray Code ordering - the same way virtual processors are numbered by Dataparallel C.

	0	1	2	3	4	5	6	7
	p0	p1	p3	p2	p6	p7	p5	p4
row 3:	6	6	7	7	8	8	9	9
row 2:	5	-1	-1	-1	-1	-1	-1	-1
row 1:	4	4	4	4	4	4	4	4
row 0:	0	0	1	1	2	2	3	3

Figure 4-5. Compiler feedback - scheduling

4.4.2 Dependency Flowgraph Output

By allowing the user to specify dependencies through variables, we have alleviated the cumbersome notion of ports and channels used in previous languages. One drawback to this new system is that mistakes may easily go unnoticed. For example, if the programmer forgets to specify an IN variable for a certain OUT, then the compiler will consider it a dangling output. No error or warning is emitted.

To alleviate the problem, we have added an extra option to the compiler, `-vcg`, which causes it to emit a file that can be read by `xvcg` ([12]). `Xvcg` allows visualization of graphs, and allows the user to see how the compiler interpreted the dependency specifications. The `xvcg` tool is publicly available via anonymous ftp to `ftp.cs.uni-sb.de` in the directory `/pub/graphics/vcg`. An example is given in Figure 4-6. The nodes in the graph represent statements. The “DFS” numbering represents the Depth First Search number given to the statement (for statement prioritizing and cycle detection purposes). The “ST” value represents the statement’s order within the `psections` block (numbered from top to bottom). The edges in the graph represent the data dependencies. By selecting “edge labels” from the `xvcg` main menu, each edge becomes decorated with its corresponding variable name (not shown).

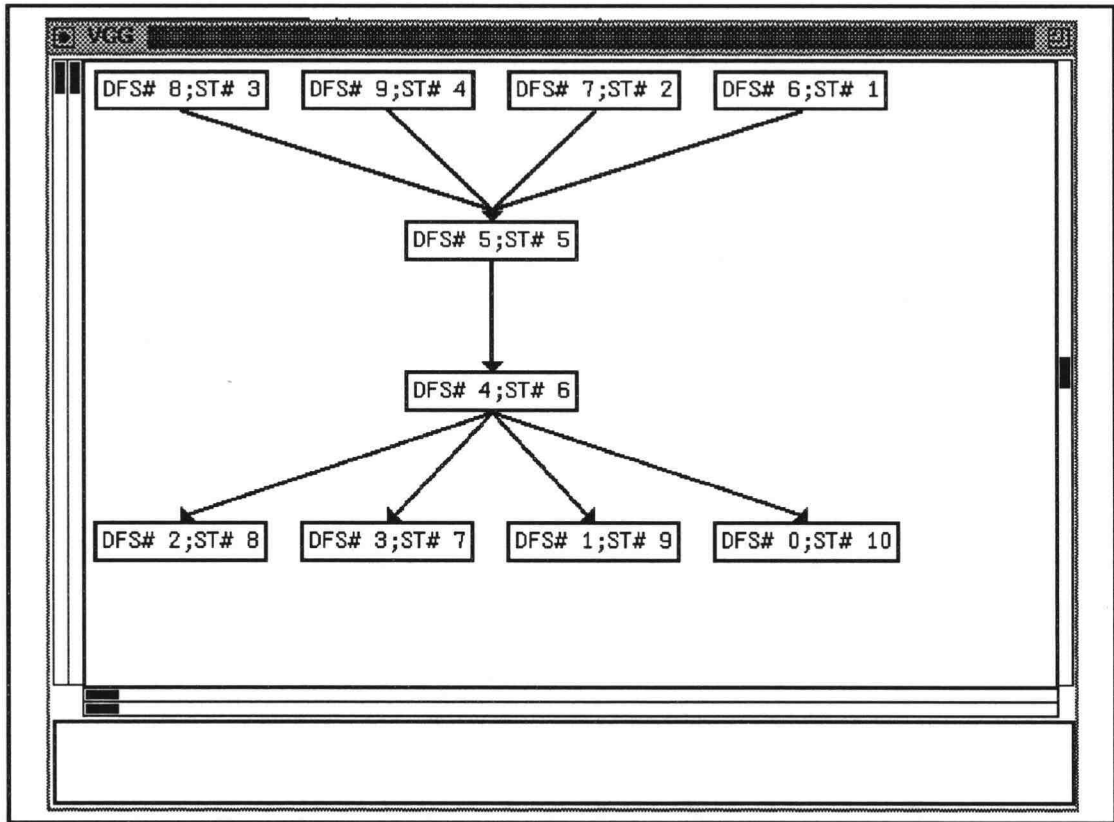


Figure 4-6. Example dependency graph visualization with xvccg

4.5 Summary

By using the `psections` statement, the programmer can allow statements inside the block an opportunity to be executed concurrently. The number of processors allocated to each statement is specified with the `ON` statement. Dependencies determine the correct order of execution of the statements and are specified with variables in the `IN` and `OUT` statements for each statement inside the `psections`.

CHAPTER 5

LANGUAGE IMPLEMENTATION

This chapter summarizes some of the implementation details for extending Dataparallel C. It also illustrates design decisions and provides justifications for some of the decisions.

The Dataparallel C language implementation is divided into two parts: the compiler and the communication library. The code generated by the compiler makes calls to the library routines for communication purposes. This isolates the compiler from modifications between target architectures. Only the communication libraries need to be updated to use the architecture's communication primitives (i.e. message send and receive function calls). The first section in this chapter summarizes the changes to the communication libraries and all remaining sections discuss compiler modifications.

5.1 Communication Libraries

The communication library is made up of several primitive operations invoked by the compiler output program. Each library was coded with the assumption that all processors contribute to communication at the same time (which supports the as-if-serial rule and the all-or-one rule). Another assumption is that the number of available processors is a power of two.

5.1.1 Eliminating the First Assumption

To efficiently implement concurrent data-parallel tasks, the first assumption must be discarded. Therefore, the library routines need to access some run-time information to determine exactly how many processors are in the subset at any time (since the size and membership of subsets changes during execution). Consequently, the new compiler

generates static lookup tables along with the output program (described below). These tables are accessed by the communication routines so that every processor can determine important information used in performing the communication algorithm:

- The node's local number in the subset
- The numbers of its predecessor and successor nodes
- The size (dimension) of the current subset

Along with the global lookup tables for this information, the compiler maintains a global flag to indicate when the program has entered a `psections` statement. Whenever the flag is not set, the communication primitives function exactly as before. Only when the flag is set do the communication primitives access the information in the tables and perform communication locally within each processor subset.

5.1.2 The Second Assumption

The second assumption maintained by the communication libraries is that the number of processors in the system is always a power of two¹. We do not find this assumption to be restrictive in any way, so it was not eliminated. It means that the current algorithms have all been reused in their current implementations. This is why the value passed to the `ON` statement must be a power of two. The feature also has advantages in scheduling and communication, since all subsets know that other subsets must also be a power of two in size.

1. This assumption originally comes from hypercube implementations, and has been maintained for universality and efficiency purposes.

5.2 Statement Scanning and Parsing

5.2.1 New Tokens

The new tokens added to the `lex` input program are:

- `psections`
- `IN`
- `OUT`
- `ON`
- `TIME`

These tokens are reserved by the compiler and cannot be used as variables, labels, function names, etc. Explanation of the `TIME` directive can be found in “Scheduling Heuristics” on page 69.

5.2.2 New Productions

Figure 5-1 shows the new or modified `yacc` productions used to scan the language extension semantics. Each of these productions creates a node for insertion into the parse tree. The first production simply creates a statement node like all other statements, except that a new tag is used to identify the `psections` statement. The second production creates a parse tree node labelled as a `psections`, and adds the compound statement as a child. The third production creates directives nodes, but not before doing some error checking. The `IN` and `OUT` list variables are type checked to ensure that each of them are mono variables. The `ON` value is checked to make sure the value is a power of two. All of the recorded tree information is used during code generation. This is described in later sections.

```
statement ::  
    psections_statement |  
    psections_sub_statement | ...  
  
psections_statement ::  
    PSECTIONS compound_statement  
  
psections_sub_statement ::  
    IN argument_expression_list ;      |  
    OUT argument_expression_list ;    |  
    ON constant_expression ;          |  
    TIME constant_expression ;
```

Figure 5-1. Language extension productions

5.3 Example Source Code

The remainder of the chapter uses a simple example to illustrate the various steps in compilation. Pseudocode for this example is given in Figure 5-2.


```

main() {
    ...
    psections {
        OUT a,b;
        ON 4;
        task5();

        IN b;
        OUT d;
        ON 2;
        task4();

        IN a;
        OUT c;
        ON 2;
        task3();

        IN c;
        OUT e;
        ON 4;
        task2();

        IN c,d;
        OUT f;
        ON 8;
        task1();

        IN e,f;
        ON 4;
        task0();
    }
    ...
}

```

Figure 5-2. Pseudocode for examples in this chapter

5.4 Collecting Information

During code generation, when a `psections` statement is encountered, a special parse table is created (called the `psections` parse table). Every entry in the table corresponds to one top-level statement in the `psections` block. Each statement is assigned a number corresponding to its order inside the block. While walking the children of the `psections` node, the `IN`, `OUT`, and `ON` information (if present) is collected and entered into the table for each statement. Every table entry contains a list of *input* and *output ports*, one for each variable in an `IN` or `OUT` statement, respectively. Every entry also contains a pointer to the parse tree for the corresponding statement. This table is used throughout code generation, and will be described in more detail below.

5.5 Creating a Flowgraph

The dependency flowgraph is created by traversing every output port in every psections table entry. For each output port, all of the input ports are scanned to find a mate with the same variable name. Any output ports without a mate are considered dangling outputs (described earlier). Any unconnected input ports at the end of the connection algorithm are considered dangling inputs, and are ignored. If an output port encounters more than one viable input port, it is cloned and connected so that each output port only has one input mate. If an output port discovers a mate that is already connected to another output port, then an error is emitted because this case illustrates the ambiguity problem discussed in Chapter 4 (see “Data Dependency” on page 25).

5.6 DFS and Topological Sort

Now that the flowgraph exists, a Depth First Search² (DFS) is performed to impose a more meaningful numbering scheme on the nodes. The deepest nodes in the flowgraph (those with no connected output ports) are assigned the lower numbers. The nodes at the next lowest level are then numbered, until the top is reached. The top-most nodes (with no connected input ports) have the largest numbers.

By assigning this numbering scheme to the nodes, a topological ordering is imposed on the nodes in the flowgraph. Once again, every node in the flowgraph (a psections table entry) is visited and every connected output port compares its DFS number with that of its mate’s DFS node number. If the mate’s DFS node number is greater than or equal to the output port’s node number, a cycle exists in the graph and an error message is emitted.

2. The DFS algorithm was adopted from [3] p.478.

5.7 Scheduling

Now that the statements and their imposed order is known, we can schedule the tasks onto the available processors. Using the recorded ON value, the scheduling algorithm can determine how many processors it must reserve for each statement. The order in which the nodes are scheduled is identical to the DFS numbering imposed earlier. Therefore, the schedule is filled in from the bottom; the first node to be scheduled is the last one to execute (because it has the lowest DFS number). The algorithm finds the lowest level in the schedule where enough processors are available to execute the task being scheduled. This lowest slot must also be above the level where any children were scheduled (to maintain the correct order of execution). An illustration (using the code from Figure 5-2) is shown in Figure 5-3. Notice that the task numbers correspond to their DFS number.

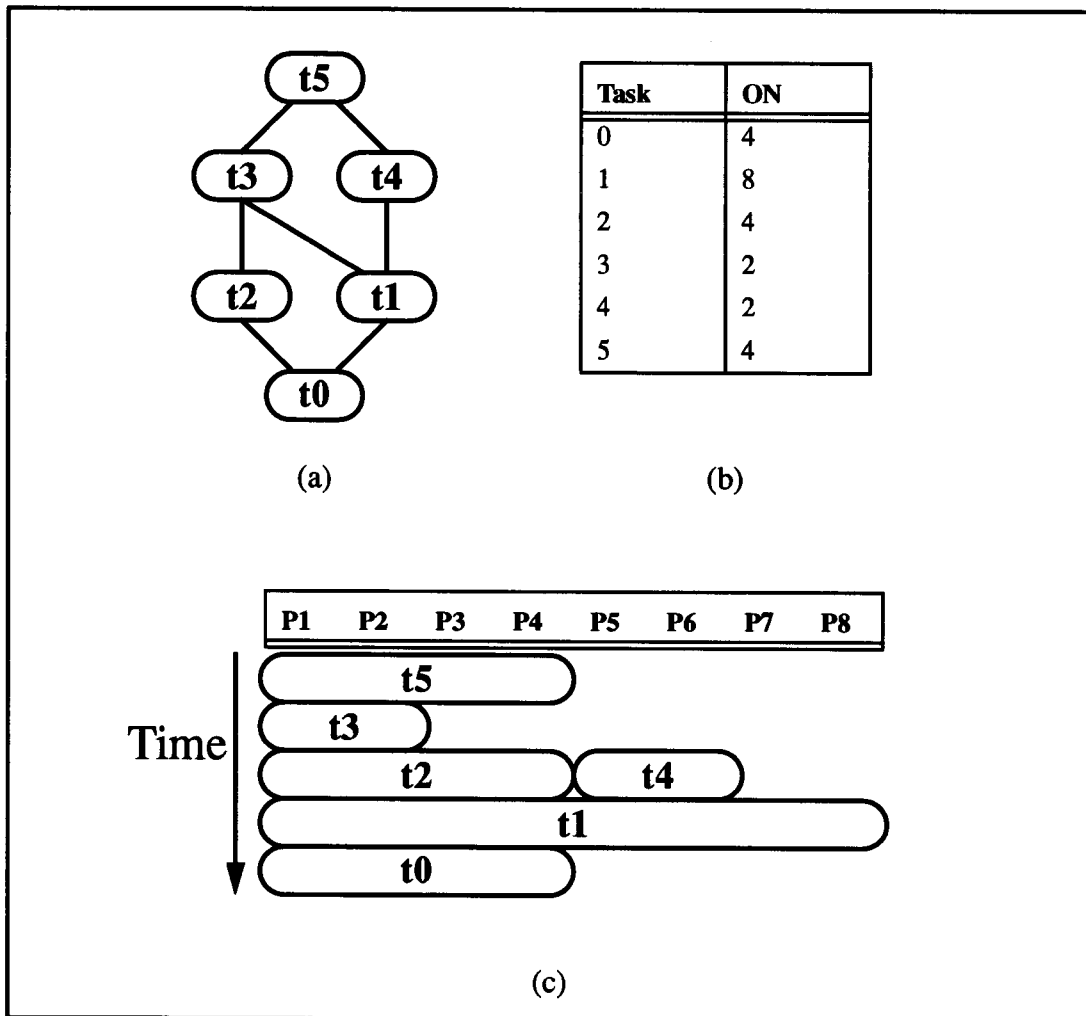


Figure 5-3. Scheduling illustration
 (a) Flowgraph and DFS numbering; (b) ON information; (c) A possible schedule generated from (a) and (b)

The algorithm described above is sufficient to create a correct schedule. The algorithm is only a heuristic, so the resulting schedule is not guaranteed to be optimal (i.e. have the lowest possible execution time). In order to guarantee an optimal schedule, an exponential time algorithm must be used. This is because the scheduling problem presented here is NP-Complete (proof given below). An exponential time algorithm is not practical for this purpose, since minor increases in the number of tasks to be scheduled will cause a drastic increase in execution time. The compiler is no longer useful if the user

must wait days, months, or years for an executable to be generated. After all, they probably could hand-code the application faster in that amount of time!

We will now prove that scheduling of data-parallel tasks is NP-Complete. We will call the problem DATA-PARALLEL TASK SCHEDULING (DPTS) since each of the tasks is a data-parallel operation that uses 1 or more processors. Two things must be shown to constitute an NP-Complete problem:

1. The problem is in NP (show $DPTS \in NP$)
2. The problem is polynomial-time reducible to some other problem known to be NP-Complete (show $PCS \leq_p DPTS$).

The problem we will reduce to is Precedence Constrained Scheduling (PCS) found in Garey & Johnson [8] p. 239. To begin, we must provide a general instance of the problem and phrase the problem in an equivalent yes/no question (decision problem).

INSTANCE: A number of processors $m \in \mathbb{Z}^+$, a set T of tasks t , each having length $l(t)=1$, each having a width $w \in \mathbb{Z}^+$ such that $w > 0$ and $w \leq m$, a partial order \Leftarrow_p on T , and a deadline $D \in \mathbb{Z}^+$.

QUESTION: Is there an m processor schedule σ for T that meets the overall deadline D , obeys the precedence constraints (i.e. such that $t \Leftarrow_p t'$ implies $\sigma(t') \geq \sigma(t) + l(t) = \sigma(t) + 1$), and every task t is allocated w consecutive processors?

The INSTANCE of the problem provides the necessary building blocks:

- The number of processors (m)
- The tasks (t), which are the statements inside the `psections` block
- The time to execute each task is assumed to be identical ($l(t)=1$).
- The number of processors requested for each statement (w)
- The partial order induced by the dependency analysis (\Leftarrow_p)

The QUESTION then asks if the schedule can be created within a deadline. This represents the notion of finding an optimal schedule, where the deadline is as low as

possible. Minimization of the deadline is no longer a decision problem, and would involve an NP-Hard proof (not provided here).

Proof for part 1: To show that $DPTS \in NP$, we need to verify a solution to the problem (not solve the problem) in polynomial time. Therefore, given any schedule, we need to find a processor with the highest (latest scheduled) task t , and make sure that $\sigma(t)+1 < D$. This verification algorithm is clearly $O(p(n))$. Therefore, $DPTS \in NP$ ♦

Proof for part 2: To show $PCS \leq_p DPTS$, we show that DPTS can emulate PCS in polynomial time. This is true because PCS is merely a restricted case of DPTS. When $w=1$ for all $t \in T$ ($\text{ON } 1$) then DPTS can solve any PCS problem. Clearly, DPTS can emulate PCS in $O(p(n))$ time. Therefore, $PCS \leq_p DPTS$ ♦

As a result of these informal proofs, DPTS is an NP-Complete problem. This means that there is no polynomial time algorithm that can solve the problem (unless $P = NP$).

5.8 Code Generation

Once scheduling is complete, the compiler can generate the static tables (described above), the code to assign different processors to different statements, and some global variables to manage the psections tables.

5.8.1 Static Tables

The static tables are generated into a separate header file which is included in the generated main program. The table is generated directly from the schedule. For every row in the schedule, the table has a smaller table that contains a mapping from every real processor to a local processor number and local size. Figure 5-4 shows the static table

information for the first three rows corresponding to schedule in Figure 5-3 (c). Any unscheduled slots treat the processors as subsets of size 1.

P	row 1		row 2		row 3	
	num	size	num	size	num	size
1	1	4	1	2	1	4
2	2	4	2	2	2	4
3	3	4	1	1	3	4
4	4	4	1	1	4	4
5	1	1	1	1	1	2
6	1	1	1	1	2	2
7	1	1	1	1	1	1
8	1	1	1	1	1	1




Figure 5-4. Example static table information

5.8.2 Assigning Processors

When a `psections` is encountered in the main program, the processors must be able to go their separate ways and work on different pieces of the program. Therefore, every row in the schedule causes the generation of the five parts listed below. Figure 5-5 shows an example of this generated code corresponding to the examples used above (Figure 5.3(c) third row). Not all of these parts are in the example because of communication optimization discussed below. The five parts generated are:

1. An update of the global static table pointer
2. A switch / case statement to divert processors to the right code
3. Communication for any incoming values

4. The actual statement

5. Communication to send any outgoing values

```

...
PsectionsTablePtr=
    PsectionsTable[2];    ➤ update static table ptr
switch(DPC_nodenum) {    ➤ divide the processors
case 0:
case 1:
case 2:
case 3:
{
    broadcast(...);      ➤ no receive, only broadcast
    task2();              ➤ perform task t2
}                        ➤ no send is necessary
break;

case 4:
case 5:
{
    if (DPC_nodenum == 4) ➤ only the leader
        receive_message(...); ➤ receives input from task 5
    broadcast(...);      ➤ distribute it locally
    task4();              ➤ perform task t4
    if (DPC_nodenum == 4) ➤ only the leader
        send_message(...); ➤ send the result to task 1
}
break;

default:                ➤ procs 6 & 7 do nothing
}                        ➤ end switch
...

```

Figure 5-5. Simplified example of generated code

The first action is to update a global pointer into the static table. Any communication primitive calls made by the tasks in this schedule row will need the correct subset information. All the needed information is given in the sub-table for this schedule row (Figure 5-4).

The processors must now be diverted to their appropriate code. The same program is running on all processors in the system. Switching on the actual processor number, groups of case statements divide the processors into their subsets to execute their separate tasks.

Once the appropriate case has been entered, all input ports must be checked. For every mated input port (of the current task), a call to message receive is generated for the *leader-processor* only. The leader-processor is the lowest numbered processor in the subset. Then a (local) broadcast is used to update the received value to all of the processors in the subset.

The statement is now generated. This is done by unparsing the users code as if it were outside the psections.

Finally, once the execution of the statement has completed, the output ports must be checked. For every mated output port (of the current task), a message send is generated by the leader-process.

By traversing every row in the schedule (from top to bottom), almost all of the code will be generated when these five steps are applied to every row. The final code generation step handles any dangling outputs. For each dangling output, a broadcast is generated from the task's leader-processor to all of the other processors in the system.

5.8.3 Communication Optimization

The generated code appears to use a lot of communication which may be stifling to performance. Fortunately, much of the communication is unnecessary. The compiler can detect when communication is unnecessary and eliminate it.

Recall that values sent between tasks must be mono. This means that a copy exists on every processor in the subset. Also recall that every subset has a leader-processor. With this in mind, the following optimizations are possible:

- If the sending and receiving leader-processors are identical, no message needs to be sent.

- If the leader-processors are the same and the receiving subset size \leq sending subset size, no communication or broadcast is necessary.
- If the subset size is 1, no broadcast call needs to be made

Figure 5-5 shows that task 2 does not receive or send. This is because its only input is from task 3, which has the same leader-processor (0). Since task 3 is only allocated 2 processors while task 2 is on four, a broadcast must be done to update the value on all the processors in the task 2 subset. Task 2 has an output to task 0, but since they share a leader-processor, no message is sent.

CHAPTER 6

USING THE LANGUAGE: A SIMPLE EXAMPLE

This chapter uses a simple example to illustrate how to use the language extensions discussed in Chapter 4. The example program is a double matrix multiply where two independent matrix multiplications are performed to yield two matrix results. Relevant portions of the program are provided below while the full source code is provided in Appendix A. Even though this is a contrived toy program, the simplification is necessary to illustrate how to get performance improvement using the language extensions; a more substantial example is described in the next chapter. The remainder of this chapter explains the purely data-parallel implementation, explains the addition of task parallelism, and summarizes performance improvement.

6.1 Double Matrix Multiply Using Only Data-Parallelism

Matrix multiply is commonly implemented in a data-parallel fashion by distributing each of the rows in the first matrix and each of the columns in the second matrix to the virtual processors. The columns are communicated among the virtual processors in a neighbor-to-neighbor fashion throughout the computation. Each virtual processor will have one row of the solution when the multiplication is complete. Some code for the Dataparallel C implementation is shown in Figure 6-1. Because this is a purely data-parallel implementation, the two multiplications are done consecutively, each using all of the available processors (second member function not shown).

More details regarding data-parallel matrix multiplication may be found in [17]. Dataparallel C implementations of matrix multiply may be found in [9].

```

#define ROW (this - P)
#define SIZE 1120
domain D { /* poly variables */
    double a[SIZE];
    double b[SIZE];
    double c[SIZE];
    double sum;
    void matrix_mult1(void);
    void matrix_mult2(void);
} P[SIZE]; /* SIZE vp's */
...
void D::matrix_mult1(void) {
    int j,k;
    /* implicit [domain D] */
    for(j=0; j<SIZE; j++) {
        sum = 0.0;
        for(k=0; k<SIZE; k++)
            sum += a[k] * b[k];
        c[(j+ROW)%SIZE] = sum;
        predecessor()->b = b;
    }
}
...
main() {
    ...
    D::matrix_mult1();
    D::matrix_mult2();
    ...
}

```

Figure 6-1. Data-parallel matrix multiplication

6.2 Adding Task Parallelism

Since the two matrix multiplies are independent of each other, they can be done concurrently. Each operation uses half of the available processors to do the multiplication in a data-parallel fashion. Therefore, the virtual processors in each half must do twice as much work. Each virtual processor has two rows and computes two rows of the solution. There is a decrease in overall communication because two columns are communicated per iteration of the outer loop. Since fewer physical processors are participating in each

solution, the number of actual messages sent decreases. Because of this increase in work and decrease in communication, the performance of the application should theoretically improve.

Figure 6-2 shows some of the code for the task parallel example. By doubling the work done by each virtual processor, four solution elements are actually computed inside the inner loop. The x and y values help determine the correct location of each result.

The main program contains the `psections` statement around the two member function calls (each member function is a data-parallel operation). There are no dependencies, so no `IN` or `OUT` statements are used. This allows the compiler to schedule the calls concurrently (when enough resources are present). The “`ON PP/2`” statement instructs the compiler to use half of the available processors (assuming `PP` is defined on the compiler invocation line with a `-D` option).

```

#define ROW (this - P)
#define SIZE 1120
domain D { /* poly variables */
  double a1[SIZE], a2[SIZE];
  double b[2*SIZE];
  double c1[SIZE], c2[SIZE];
  double sum1, sum2, sum3, sum4;
  void matrix_mult1(void);
  void matrix_mult2(void);
} P[SIZE]; /* SIZE vp's */
...
void D::matrix_mult1(void) {
  int j, k, x, y;
  /* implicit [domain D] */
  for(j=0; j<SIZE/2; j++) {
    sum1 = sum2 = sum3 = sum4 = 0.0;
    for(k=0; k<SIZE; k++) {
      sum1 += a1[k] * b[k];
      sum2 += a1[k] * b[k+SIZE];
      sum3 += a2[k] * b[k];
      sum4 += a2[k] * b[k+SIZE];
    }
    x = ROW*(SIZE/2);
    y = ROW*(SIZE/2)+SIZE/2;
    c1[(j+x)*SIZE] = sum1;
    c1[(j+y)*SIZE] = sum2;
    c2[(j+x)*SIZE] = sum3;
    c2[(j+y)*SIZE] = sum4;
    predecessor()->b = b;
  }
}
...
main() {
  ...
  psections {
    ON PP/2;
    D::matrix_mult1();
    ON PP/2;
    D::matrix_mult2();
  }
  ...
}

```

Figure 6-2. Task-parallel matrix multiplication

6.3 Timing Results

Running both programs on dedicated partitions of the Meiko CS-2 supercomputer with 1120 x 1120 sized matrices of doubles yields the results in Figure 6-3. Note that this is a log-log plot and the y-axis shows speed as the inverse of execution time. The “task-parallel” curve actually represents the program which has data-parallelism nested inside task-parallelism. The “data-parallel” curve represents the purely data-parallel solution.

Clearly, the “task-parallel” curve shows better performance on 2, 4, and 8 processors. Table 1-1 shows the performance improvement of the task-parallel case over the purely data-parallel case for each of these instances.

Table 1-1. Double Matrix Multiply Improvement with Task-Parallelism

Processors	% Improvement
2	61.2%
4	65.4%
8	66.7%

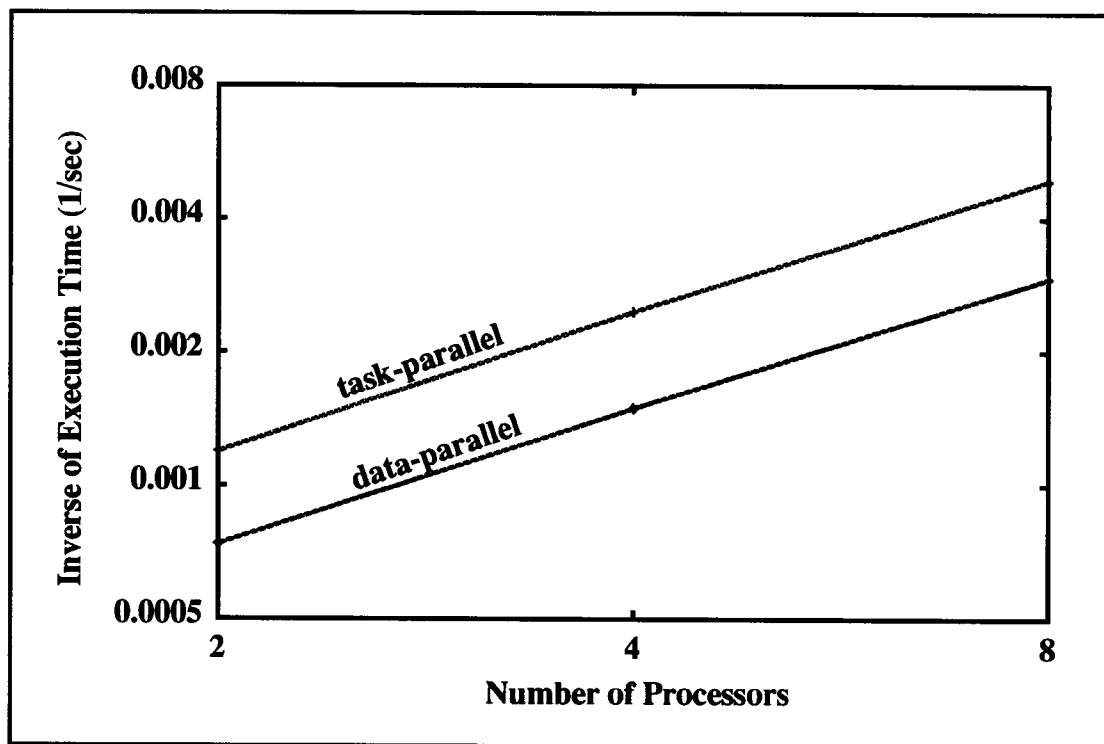


Figure 6-3. Timing results for double matrix multiply (1120 x 1120 doubles)

CHAPTER 7

BANDED SYSTEM SOLVER

The description of our parallel language extensions is not complete without implementing some real-world applications that can benefit from the language's strengths. Applications are essential to validate the language and to benchmark the generated code against other languages and programming styles. In our case, application execution times are compared against the purely data-parallel solutions. This chapter illustrates how a data-parallel application executes faster when the data-parallel subroutines can be executed concurrently.

The example given in this chapter also illustrates a path for converting Fox's Class II applications into Class IIICG-IIFG (see "Types of Parallel Applications" on page 3) via the new language. Some programs written in Dataparallel C may benefit from this migration path.

7.1 Introduction

Real-world systems are commonly modeled using complex equations to describe the systems' behavior. Once a satisfactory model is obtained, the systems may be simulated under conditions which do not (yet) exist.

Given the complexity of these models, the size of the systems to be modeled, the desired accuracy of the simulation, and the length of the simulation, it is easy to see that high performance computers should be ideally suited for the simulations. After all, the simulation is only useful if it can return desired results in a timely manner. Unfortunately, there are three issues which may confound the use of parallel computers for numerical simulation:

- Architectural adaptability
- Efficiency (performance)
- Programmability

The solution technique used for this chapter's example has been shown ([11]) to alleviate the first two problems by choosing an appropriate algorithm for the problem at hand (discussed below). The third problem can be eased by using a high-level parallel language for the implementation, although there may be a loss in efficiency.

7.2 Parallelization

The remainder of the chapter describes our results in parallelizing a solution technique for modeling ocean currents. The solution technique uses domain decomposition because of its proven usefulness in this system; Santhosh Kumaran [11] compared different parallelization techniques in a finite element regional ocean circulation model. He showed that by using domain decomposition, it is possible to develop a portable and efficient implementation of the simulation. The methodology behind Kumaran's implementation is reused in this example.

The critical portion of the implementation is the solution of a system of linear equations obtained from the domain decomposition (Figure 7-1 on page 56). The coefficient matrix has a peculiar structure, which the solution scheme exploits to improve the performance. Figure 7-1 shows this coefficient matrix (M), where all of the non-zero values lie in a "band" across the diagonal of the matrix (shaded region). Details regarding the solution scheme are given in the next section of this chapter.

Solution of the system of equations appears to be ideally suited for a data-parallel implementation. Unfortunately, this system is not efficiently solved with conventional data-parallel languages. For example, Dataparallel C maintains SIMD semantics which restrict the solution to using either one or all of the processing resources (for a detailed description of the all-or-one rule see "Resource Allocation" on page 22). This restriction hinders performance when parts of the solution show optimal performance on a subset of resources.

A solution technique which uses task parallelism to allocate smaller sets of resources to data-parallel solutions is required to achieve optimal performance. The task-parallel

framework not only limits communications to within each subset of resources, but it also allows concurrent solution of data-parallel subsets by optimally utilizing all processing resources simultaneously.

The language used to implement the combined task and data parallel solution is our extension of Dataparallel C discussed throughout this thesis. The compiler was used to generate all executables. No “hand-coding” was done. The source code is given in Appendix B.

The remainder of the chapter summarizes the solution technique and reports our findings in implementing the linear system solver. The advantages of a combined task-parallel and data-parallel solution (over the purely data-parallel solution) are expounded upon.

7.3 Solving A Banded System of Linear Equations

After collecting the data from a Finite Element Method, a system of linear equations results. Let M represent the coefficient matrix data, x represent the vector of unknowns, and b the known right-hand-side solution vector. The equation being solved is then:

$$Mx = b$$

Solving for the equations’ unknowns corresponds to one time step in the simulation. The solution vector x then becomes the b vector in the next time step. The M matrix retains the same values for every time step.

As mentioned previously, all non-zero values in M lie in a “band” across the diagonal of the matrix (Figure 7-1; shaded areas represent known non-zero values). The width of this band is the *bandwidth*, and will be denoted by B .

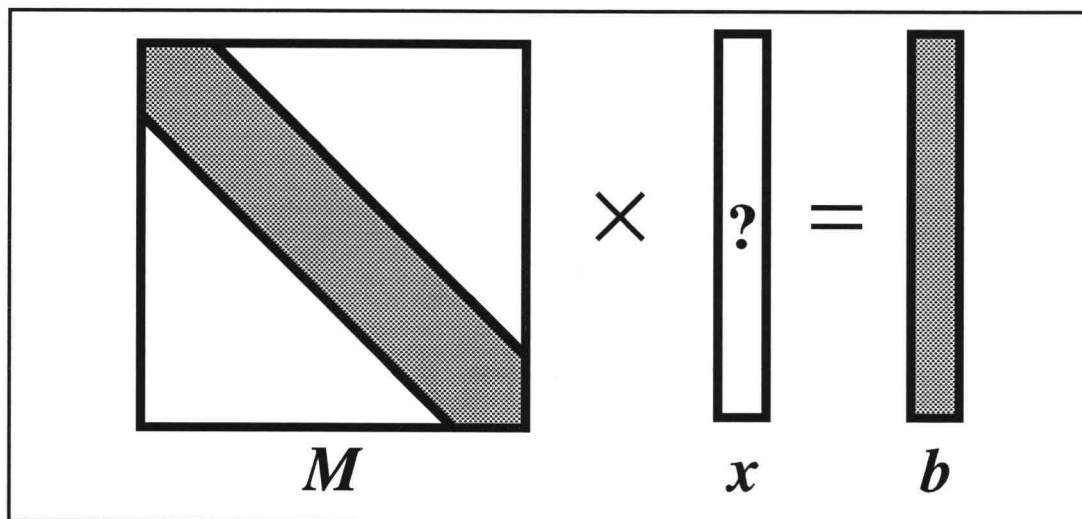


Figure 7-1. System of linear equations

The first step in solving the banded system requires cutting the matrix into partitions and rearranging the rows and columns. The width of this cut will be denoted by \mathbf{BA} and is equal to $(\mathbf{B}-1)/2$. The number of partitions induced by the cuts is \mathbf{P} , each of size \mathbf{PS} (Partition Size). The cuts are applied in even intervals across the columns of the matrix. The cut regions are all moved to the top of the matrix, with all partition rows moving downward to fill in the gaps. The values in the solution vector b are moved in a corresponding manner. Similarly, the cuts are then applied across the rows with the cut regions being moved to the left side of the matrix and all partition columns sliding to the right to fill in the gaps. A rearranged matrix with 3 partitions is shown in Figure 7-2 (shaded areas contain non-zero values), and will be denoted A . The thick horizontal and vertical divisions represent the boundary between the moved partitions and the remaining data. The dashed lines represent boundaries among the partitions and remaining data.

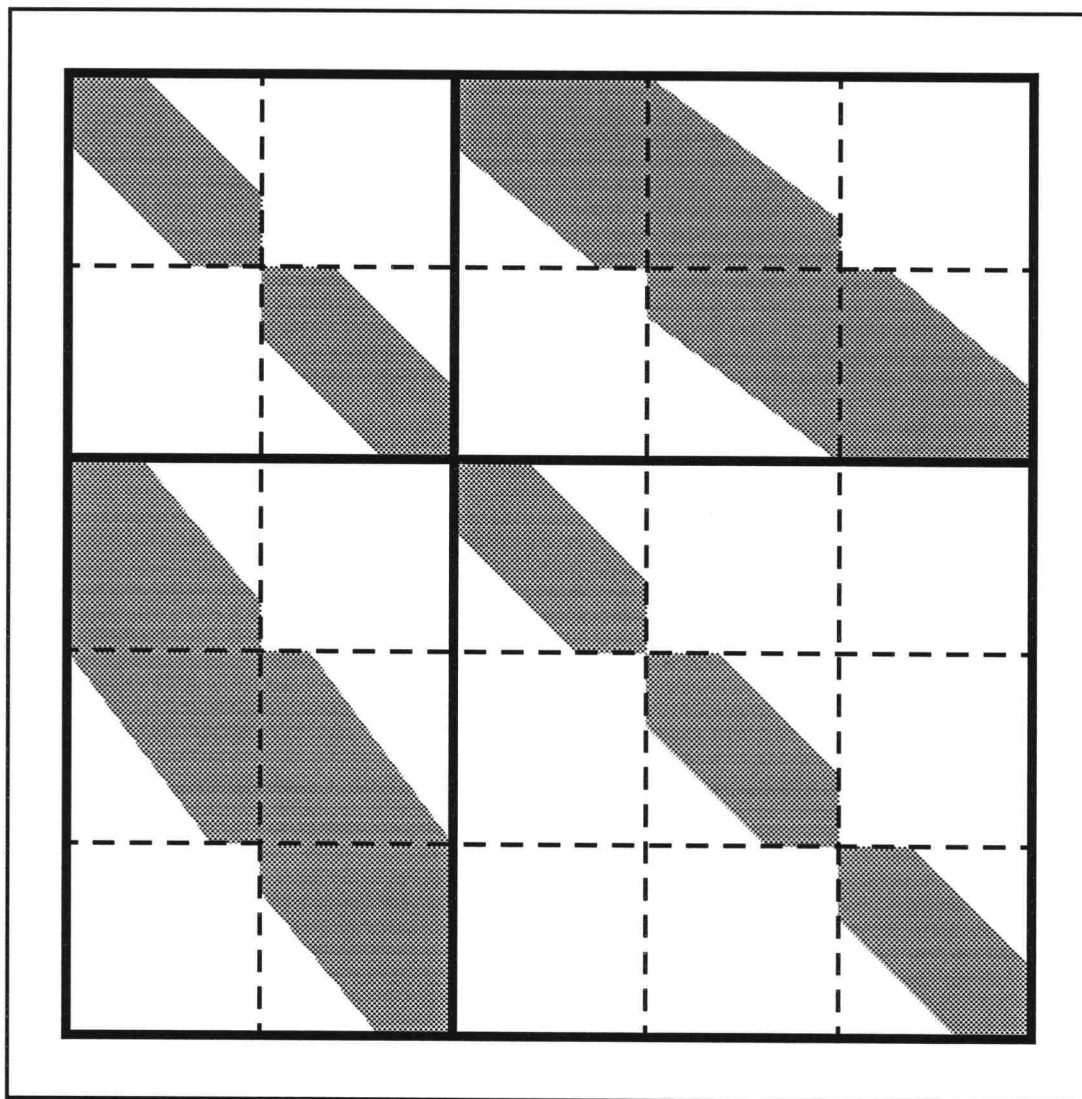


Figure 7-2. Rearranged input matrix

The area where the cuts were moved is called the *interface* portion of A . The remainder is called the *domain* part of A . The areas these names refer to depend on whether you are considering rows or columns. Once the matrix is rearranged, names are assigned to some of the sub-matrices that appear inside the rearranged matrix (those with values). These are shown in Figure 7-3, along with the naming conventions for the matrices within each sub-matrix.

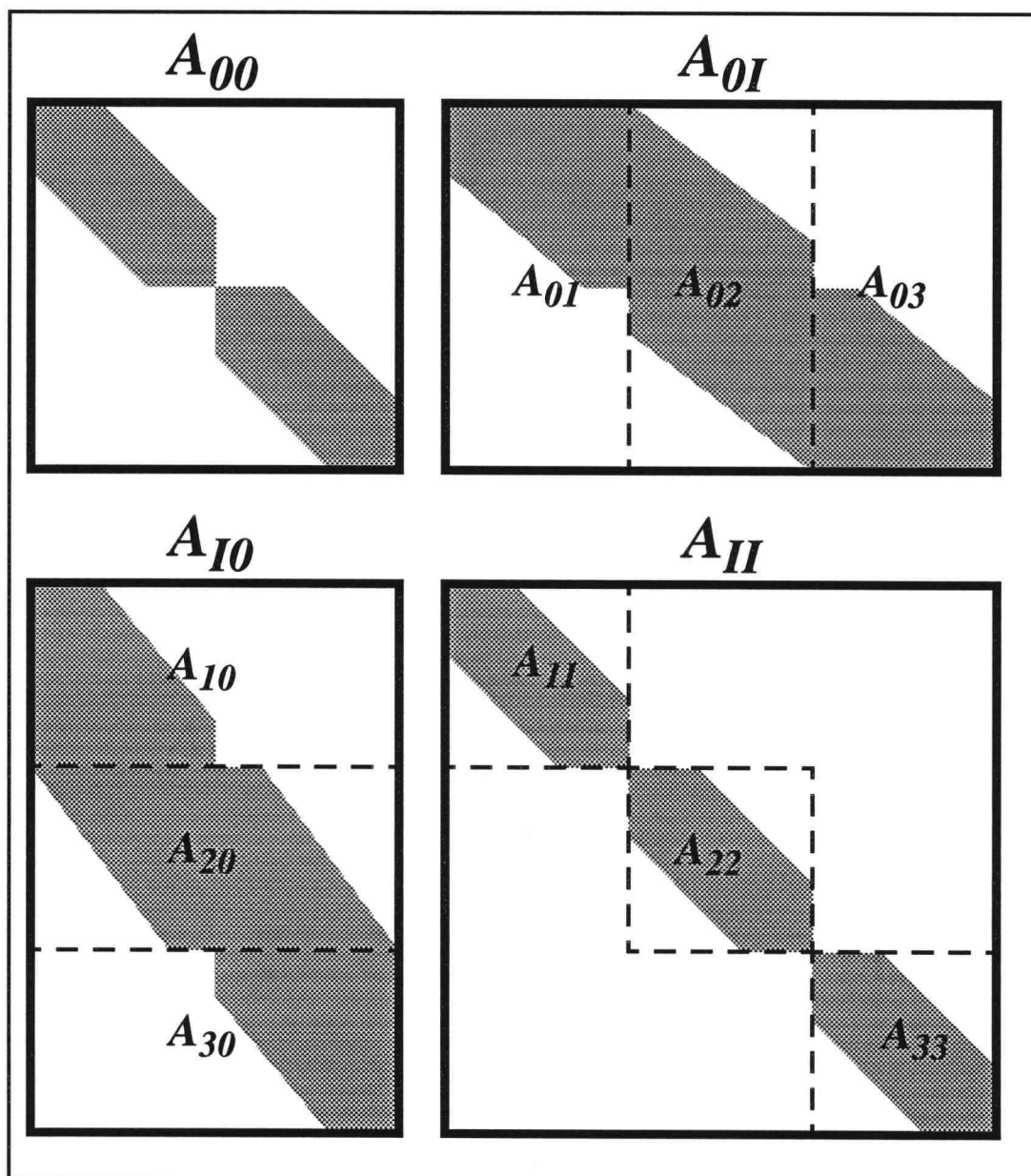


Figure 7-3. Sub-Matrices within the rearranged input matrix

Using these sub-matrix names, the equations to solve for the unknowns are given in the following equations:

$$r_o = b_o - \sum_{i=1}^P [A_{0i} (A_{ii})^{-1} b_i] \quad (\text{EQ 1})$$

$$x_o = [A_{00} - A_{0I} (A_{II})^{-1} A_{I0}]^{-1} r_o \quad (\text{EQ 2})$$

$$x_i \Big|_{i=1, 2, \dots, P} = A_{ii}^{-1} [b_i - A_{i0} x_o] \quad (\text{EQ 3})$$

where x_o and b_o are the interface portions of the x and b vectors respectively; x_i and b_i are the domain portions corresponding to each partition.

At first, it may seem that this scheme cannot possibly perform efficiently because of all the matrix inversions involved. As stated earlier, the input matrix does not change for each iteration of the simulation. Therefore, some of the values need only be computed once before the simulation begins. These computations are therefore removed from the simulation loop and only computed once:

$$A_1 = A_{0i} (A_{ii})^{-1} \quad (\text{EQ 4})$$

$$A_2 = [A_{00} - A_{0I} (A_{II})^{-1} A_{I0}]^{-1} \quad (\text{EQ 5})$$

$$A_3 = A_{ii}^{-1} \quad (\text{EQ 6})$$

Substituting Equation 4 into Equation 1, Equation 5 into Equation 2, and Equation 6 into Equation 3 finally yields the equations which are done in parallel for each simulation iteration:

$$r_o = b_o - \sum_{i=1}^P [A_1 b_i] \quad (\text{EQ 7})$$

$$x_o = A_2 r_o \quad (\text{EQ 8})$$

$$x_i \Big|_{i=1, 2, \dots, P} = A_3 [b_i - A_{i0} x_o] \quad (\text{EQ 9})$$

7.4 Implementation of the Solution Technique

Without providing pseudocode for all of the equations, we will explain the basic structure of the implementation of these equations. The two prevalent operations in Equation 7, Equation 8, and Equation 9 are matrix-vector multiplication and vector addition (subtraction). For simplification, we will focus on the multiplication because it is much more time consuming.

Matrix-vector multiplication involves 2 loops, one nested inside the other. The outer loop traverses the rows of the matrix and elements of the solution vector. The inner loop traverses the columns of the matrix and the elements of the input vector, performing a summation into the solution vector space. Recall that in Equation 7 and Equation 9, the matrix-vector multiplications are done P times, adding a third loop around the matrix vector multiplications. The pseudocode for this is shown in Equation 7-4.

```

for i ← 1 to number of partitions  $P$       ► Loop 1
  for j ← 1 to number of rows              ► Loop 2
    for k ← 1 to number of columns        ► Loop 3
       $v_{2_i}[j] \leftarrow M_i[j][k] * v_{1_i}[k]$   ► matrix  $\times$  vector

```

Figure 7-4. Pseudocode for repeated matrix-vector multiplications

Note that Equation 8 actually involves only loop 2 and loop 3 because only one matrix-vector multiplication is performed.

7.4.1 Data-Parallel Solution

The initial data-parallel implementation is created by parallelizing loop 2 in Figure 7-4. Each virtual processor is assigned one row of the matrix and one element of the solution for each of the \mathbf{P} matrix-vector multiplications. The communication involved is neighbor-write inside the inner loop.

Another way to implement a data-parallel solution is to parallelize loop 1 instead of loop 2 (Figure 7-4). Each virtual processor now computes its own matrix-vector multiply, alleviating much of the communication.

7.4.2 Augmenting the Example with Task-Parallelism

A good task-parallel solution would involve pipelining the simulation iterations and overlapping operations from different iterations. Unfortunately, this implementation would be slow because all of the previous iteration's answers are needed in Equation 7 to solve for the r_o value used in Equation 8. This bottleneck would result in an inefficient pipeline, and is not worth discussing further.

The next best way to add task-parallelism is to unroll loop 1 (the outer loop) and perform these independent operations concurrently, each on a subset of the available processors. Data parallelism is maintained by parallelizing loop 2, as in the initial data-parallel implementation discussed above. Therefore, each of the unrolled iterations is a data-parallel solution using the subset of resources assigned to it.

7.5 Summary of Findings

The graph in Figure 7-5 may look overwhelming at first, but a brief explanation will elucidate how this data shows an improvement when task parallelism is added to Dataparallel C. Note that the y-axis of the graph is expressed in speed rather than time.

The input size for this example (N) is 1008 and the data are all stored as doubles. The P and B parameters are 4 and 33 respectively. All timings were made on dedicated partitions of a 16-node MEIKO CS-2 supercomputer.

7.5.1 Purely Data-Parallel Implementation

The two dotted lines (labelled DP1 and DP2 in Figure 7-5) represent two versions of the purely data-parallel solution in Dataparallel C (ignore the other lines for now). The poorly performing version (DP2) was created by parallelizing loop 2 (in Figure 7-4) of each matrix-vector multiply, as discussed above. This may be considered a naive implementation because it is unclear that the communication involved would be so time consuming, thus hurting the performance of the application. This is why the graph for DP2 (Figure 7-5) slowly decreases with the addition of processors.

The next data-parallel implementation (DP1) is better because loop 1 was parallelized instead of loop 2 (Figure 7-4), as discussed above. The graph for DP1 (Figure 7-5) clearly shows better performance and speedup because of the decrease in communication.

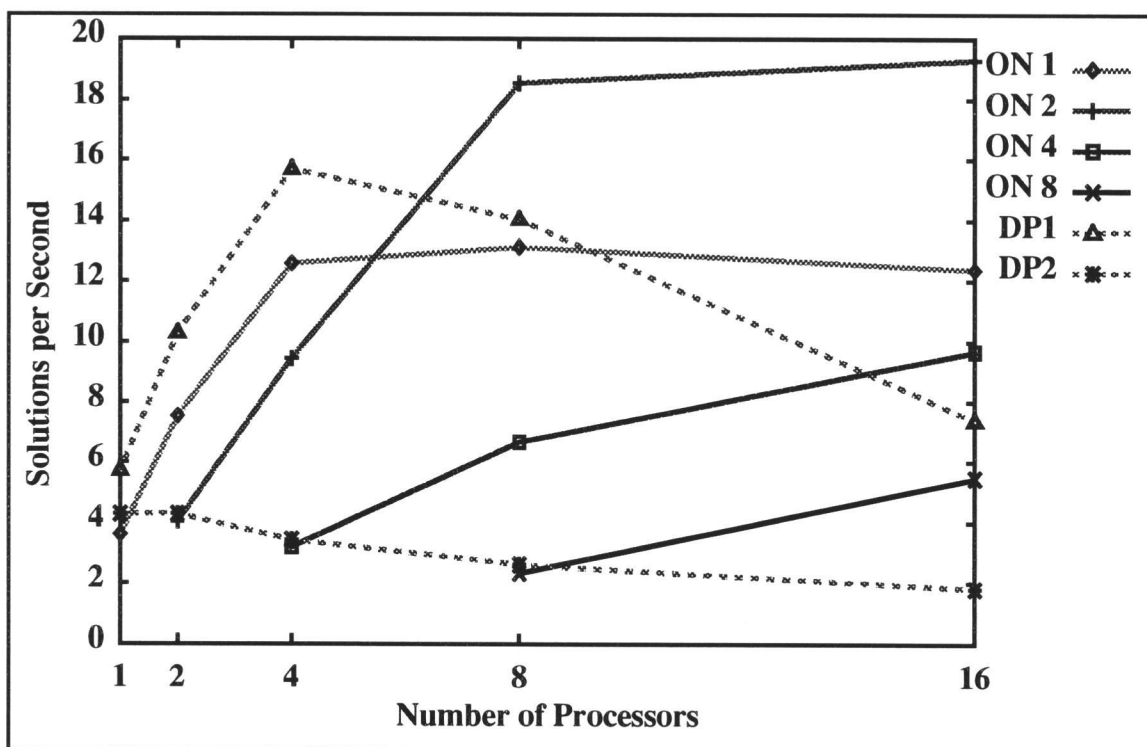


Figure 7-5. Speed vs. Processors for all implementations

Notice that the graph of DP2 shows good speedup only to 4 processors. Adding processors past that point causes the data parallel solution to drop off. There are two reasons for this drop-off:

- For this example, the number of partitions $P = 4$. Any addition of processors beyond P does not show any improvement because it is the outer loop over P that is parallelized.
- Dataparallel C always uses all processors for communication, even if no significant work is being done on some of them. This adds useless overhead, and causes performance to decrease beyond 4 processors.

The observant reader will now ask: “*Why not just use 4 processors?*” As shown in [11], the number of processors used to solve the system tends to be much greater than the number of partitions (P). Therefore, the purely data-parallel solution *under-utilizes* system resources which is wasteful. Also note that [11] shows how large increases in the input size will not cause large increases in P . Therefore, system resources will be under-utilized,

even when larger systems are input. The next section describes a different approach which allows more resources to be used in an efficient way.

7.5.2 Combined Task-Parallel and Data-Parallel Implementation

By adding task parallelism to the solution of this system, better performance is obtained by using more of the available processing resources. Dataparallel C maintains the all-or-one rule for allocating processor resources: sequential code is done on every processor; parallel code, especially communication, is divided among all of the processors (which must maintain loose synchronization with the other processors). The addition of task parallelism loosens this restriction and allows subsets of resources to be applied to simultaneous data-parallel operations. Any communication is limited to the subset and does not require any synchronization with other processors.

The solid line graphs in Figure 7-5 represent the task-parallel implementation of the system. Data parallelism is maintained by parallelizing loop 2 (as in DP2), and task parallelism is obtained by unrolling loop 1 (the outermost loop) and assigning a subset of resources to the solution of each of these unrolled iterations. Each of the unrolled iterations is a data-parallel solution using the subset of resources assigned to it. The number of processors assigned to a subset is depicted by the ON value in Figure 7-4. Therefore, the best solution for this input size is actually obtained by executing on 8 or more processors, with 2 processors allocated to each of the $P=4$ data-parallel subsets (ON 2).

The observant reader will notice that the ON 4 and ON 8 cases have disappointing speedup. This can be attributed to the following reasons:

- Adding more processors to a data-parallel subset will eventually show a decrease in performance when less work, and more communication, is done per processor (just as in a purely data-parallel application). Hence the slope of these curves is not as steep as in the ON 1 and ON 2 cases.

- The total number of processors needed by the ON 4 and ON 8 cases is 16 ($P=4 * ON\ 4$) and 32 ($P=4 * ON\ 8$) respectively. Whenever these cases are compiled onto fewer than this number of processors, the compiler overlaps the task-parallel work being done (i.e. the subsets take turns using the resources that are available when not enough are available for total concurrency). This overlapping causes the overall performance to degrade below that of the purely data-parallel version, unless enough processors are used so that overlapping does not occur. Notice that the ON 4 case is faster than the purely data-parallel version only when the number of processors is 16.

The observant reader will also notice that the ON 1 case is simply a data-parallel solution. By allowing only one processor to solve each subset, we are modelling the data-parallel solution (DP1) because in both implementations, each processor is solving one of the P outer loop iterations. This is possible because data parallelism is a special case of task parallelism. Unfortunately, the task-parallel model (ON 1 curve) mimics, but does not equal the data-parallel solution (DP1). The gap between the two curves is a result of extra overhead incurred by the calls to communication routines in the task-parallel solution (which are not present in the data-parallel solution). Even though no messages are actually sent, communication among virtual processors on each single processor is implemented with `memcpy` calls. The data-parallel implementation does not have any communication among the virtual processors in the inner loop, and does not suffer from this overhead.

Now notice that beyond 8 processors, the ON 1 case actually *does* outperform the data-parallel case. As mentioned earlier, these instances do not use all of the resources because the number of processors is greater than P . The difference is that the purely data-parallel version still uses all of the processors for communication, and the overhead degrades performance. Conversely, the task-parallel implementation does not need to use the processors, so that any extra processors sit idle and don't interfere with computation and communication. This explains why the ON 1 graph flattens off after 4 processors, and the ON 2 case flattens off after 8 processors. The addition of processors past these points does not provide any extra computation power.

7.6 Chapter Summary

The purely data-parallel implementation of the banded linear system solver is improved by adding task parallelism. Improvements result from:

- More efficient use of available resources
- Concurrency between operations
- Independent concurrent operations
- No interference of unused resources with utilized resources

CHAPTER 8

CONCLUSION

8.1 Conclusion

How can the usefulness and completeness of our work be determined? One of the renowned masters of programming languages has provided some insight. The following quote is taken from “Hints on Programming Language Design” by C.A.R. Hoare presented at POPL 1973 ([10] p. 39):

The designer of a new feature should concentrate on one feature at a time. If necessary, he should design it in the context of some well known programming language which he likes. He should make sure that his feature mitigates some disadvantage or remedies some incompleteness of the language, without compromising any of its existing merits. He should show how the feature can be simply and efficiently implemented. He should write a section of a user manual, explaining clearly with examples how the feature is intended to be used. He should check carefully that there are not traps lurking for the unwary user, which cannot be checked at compile time. He should write a number of example programs, evaluating all the consequences of using the feature, in comparison with its many alternatives.

The following list summarizes how we have abided by Hoare’s guidelines:

- **Concentrate on one feature:** The only feature is the addition of task parallelism to a data-parallel language.
- **Design in the context of a well know language we like:** We chose to design our language extensions in the context of Dataparallel C. The language is well known, and we like it.
- **Mitigate a disadvantage:** The as-if-serial rule and the all-or-one rule of processor allocation are overly restrictive and do not allow optimal performance for some applications. By loosening these rules, we have extended Dataparallel C to allow task parallelism.

- **Don't compromise the language's existing merits:** None of the existing merits of Dataparallel C were compromised. Data-parallel programming is still supported and the extended language can still compile Dataparallel C programs.
- **Show implementation:** Chapter 5 provides an overview of our implementation.
- **Write a user manual:** Chapter 4 provides an explanation of the language features and Chapter 6 provides a simple example.
- **Eliminate lurking traps:** The compiler provides feedback to the user describing what it is doing during dependency analysis and scheduling. These are possible areas where the user can make a mistake without the compiler being able to catch it.
- **Write example programs:** Chapter 6 provides a simple program, while Chapter 7 shows the implementation of a real simulation program. The programs were compared with their alternatives - pure data-parallelism.

8.2 Future Work

This section lists some ideas for future work which may be interesting for future Masters or Ph.D. topics. The ideas provided here are intended to expand upon the work we have already done.

8.2.1 Dynamic Virtual Processors.

Investigating dynamic virtual processors (possibly in C*) may help alleviate the coding difficulties described earlier. By assigning different numbers of virtual processors (instead of physical processors) to tasks, any data-parallel code could run on any subset of processors without modification. The investigator needs to be wary of the following issues:

- Full concurrency must be maintained between all of the processors. All of the virtual processors being emulated by a processor must be assigned to the same task.

- Proper allocation of memory for all poly variables residing on each virtual processor. The worst case is when one processor must emulate all of the virtual processors in a domain because all poly variables will reside in one physical memory as if they were mono. The space advantage of poly variables would be lost in this case.

8.2.2 Scheduling Heuristics

We have already added a fourth statement to our set of language extensions. The `TIME` statement accepts an integer value which is currently ignored. The value is a hint by the user as to how long the current task will take relative to the other tasks. This value can be used by the scheduler heuristic to possibly create a better schedule ([13], [18]).

Another improvement to the scheduler would take into account message sizes. Tasks sharing lengthy communications should be scheduled as close to each other as possible. If they are scheduled onto the same processor, the communication can be eliminated entirely. Elimination of costly communication should cause significant increases in performance.

8.2.3 Automatic Data Dependency Analysis

If a compiler can analyze the data dependencies between tasks of a psections, then the `IN` and `OUT` statements can be eliminated. Data flow analysis can reveal which tasks must wait on others, and the variables which must be communicated between them. Programming with our language extensions would be simplified.

8.2.4 Nesting

The use of data-parallel and task-parallel library routines would help parallel programmers by providing a simple method of code reuse. The algorithms inside the

libraries would also be optimized for the best possible performance. Providing our language with the ability to arbitrarily nest data-parallel and task-parallel modules within each other will facilitate the use of parallel libraries.

8.2.5 Run-time Feedback

Jaspal Subhlok (Carnegie Mellon) has proposed and implemented an efficient method of automatically mapping tasks to processors [22]. His compiler executes the program with various mappings, collects run-time information, and determines an optimal mapping of the tasks to processors. If dynamic virtual processors are implemented (as discussed above), a similar mapping scheme may be used. This would eliminate the need for the `ON` statement.

8.2.6 Automatic Task Compilation

If the feedback system were combined with automatic dependency analysis (both are discussed above), the need for any extra compiler directives or statements would be eliminated. The `psections` statement is the only extension left. With so much automatic analysis, the `psections` statement might also be removed. The compiler could assume that all the code is inside a `psections`, and “do the right thing” with these smart modifications. This means that all Dataparallel C programs would be automatically amenable to task parallelism without any modification by the programmer.

BIBLIOGRAPHY

- [1] Henri Bal and Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994.
- [2] Mani Chandy and Ian Foster. Integrated Support for Task and Data Parallelism. California Institute of Technology & Argonne National Lab, August 27, 1993. To appear in *International Journal of Supercomputer Applications*.
- [3] Cormen, Leiserson, Rivest. *Introduction to Algorithms*. Massachusetts Institute of Technology, 1990.
- [4] Ian Foster and Mani Chandy. *FORTRAN M: A Language for Modular Parallel Programming*. Argonne National Laboratory, October 1992.
- [5] Ian Foster and Carl Kesselman. Integrating Task and Data Parallelism. In *Proceedings of Supercomputing '93*, (Portland, OR., November 1993).
- [6] Ian Foster, Ming Xu, Bhaven Avalani, Alok Choudhary. *A Compilation System That Integrates High Performance Fortran and Fortran M*. Argonne National Lab and Syracuse University, to appear in *Proc. 1994 Scalable High Performance Computing Conf.*, IEEE Computer Science Press.
- [7] Geoffrey Fox. *The Architecture of Problems and Parallel Portable Software Systems*. Tech Rep. CRPC-TR91172, Northeast Parallel Architectures Center, 1991.
- [8] Garey & Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness*. Bell Telephone Laboratories Inc., 1979.
- [9] Philip Hatcher and Michael Quinn. *Data-Parallel Programming on MIMD Computers*. Massachusetts Institute of Technology, 1991.
- [10] E. Horowitz. *Programming Languages: A Grand Tour*. Computer Science Press Inc., Second Edition, 1985.
- [11] Santhosh Kumaran and Robert Miller. A Comparison of Parallelization Techniques for Finite-Element Modelling of Quasi-Geostrophic Circulation. Submitted to *International Journal of Supercomputer Applications and High Performance Computing*.

- [12] Iris Lemke and Georg Sander. *Visualization of Compiler Graphs Design Report and Documentation*. Universität des Saarlandes and the COMPARE Consortium, 1994.
- [13] Richard Maliszewski and James Reinders. Scheduling Task Parallel Programs Using VLIW Techniques. *Proceedings of the Intel Software Development Conference*. October 10-12, 1994.
- [14] Cherri Pancake. Software Support for Parallel Computing: Where Are We Headed? *Communications of the ACM*. November 1991, Volume 34, No. 11, pp.53-64.
- [15] Cherri Pancake. Languages for High-Performance Computing: A Smorgasbord? *IEEE Parallel and Distributed Technology (Systems & Applications)*. February 1993, pp.68-72.
- [16] Cherri Pancake. The Changing Face of Supercomputing. *IEEE Parallel and Distributed Technology (Systems & Applications)*. November 1993 pp.12-15.
- [17] Michael Quinn. *Parallel Computing Theory and Practice*. McGraw-Hill Inc., 1994.
- [18] James Reinders and Damien Macielinski. Application Oriented Parallel Compilers. *Proceedings of the Intel Software Development Conference*. September 8-10, 1993.
- [19] Brad SeEVERS, Michael Quinn, and Philip Hatcher. A Parallel Programming Environment Supporting Multiple Data-Parallel Modules. *International Journal of Parallel Programming*. Vol. 21, No. 5, 1992.
- [20] Jaspal Subhlok, James Stichnoth, David O'Hallaron, Thomas Gross. Exploiting Task and Data Parallelism on a Multicomputer. Carnegie Mellon University. In the *Fourth ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*. May 1993, San Diego, CA.
- [21] Jaspal Subhlok and Thomas Gross. *Task Parallel Programming in FX*. Technical Report CMU-CS-94-112 (Preliminary Version), School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.
- [22] Jaspal Subhlok. *Automatic Mapping of Task and Data Parallel Programs for Efficient Execution on Multicomputers*. November 1993, CMU-CS-93-112, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

- [23] Jaspal Subhlok, David O'Hallaron, Thomas Gross, Peter Dinda, and Jon Webb. *Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs*. January 1994, CMU-CS-94-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.
- [24] Thinking Machines Corporation. *C* Programming Guide*. August 1990.
- [25] *X3H5 Parallel Extensions for Programming Language C*. Document number X3H5/93-SD3, Revision E, October 26th 1993.

APPENDICES

APPENDIX A

Source Code for Double Matrix Multiply

```

#include <stdio.h>
#define ID (this - P)
#define ROW (this - P)
#define SIZE 1120

domain D {
    double a1[SIZE];
    double a2[SIZE];
    double b[2*SIZE];
    double c1[SIZE];
    double c2[SIZE];
    double d1[SIZE];
    double d2[SIZE];
    double sum1, sum2, sum3, sum4;
    void matrix_mult1(void);
    void matrix_mult2(void);
    void Init(void);
} P[SIZE];

void D::matrix_mult1(void)
{
    int j;
    int x,y;
    for(j=0; j<SIZE/2; j++) {
        int k;
        sum1 = sum2 = 0.0;
        for(k=0; k<SIZE; k++) {
            sum1 += a1[k] * b[k];
            sum2 += a1[k] * b[k+SIZE];
            sum3 += a2[k] * b[k];
            sum4 += a2[k] * b[k+SIZE];
        }
        x = ROW*(SIZE/2);
        y = ROW*(SIZE/2)+SIZE/2;
        c1[(j+x)%SIZE] = sum1;
        c1[(j+y)%SIZE] = sum2;
        c2[(j+x)%SIZE] = sum3;
        c2[(j+y)%SIZE] = sum4;
        predecessor()->b = b;
    } /* for j */
}

void D::matrix_mult2(void)
{

```

```

int j;
int x,y;
for(j=0; j<SIZE/2; j++) {
    int k;
    sum1 = sum2 = 0.0;
    for(k=0; k<SIZE; k++) {
        sum1 += a1[k] * b[k];
        sum2 += a1[k] * b[k+SIZE];
        sum3 += a2[k] * b[k];
        sum4 += a2[k] * b[k+SIZE];
    }
    x = ROW%(SIZE/2);
    y = ROW%(SIZE/2)+SIZE/2;
    d1[(j+x)%SIZE] = sum1;
    d1[(j+y)%SIZE] = sum2;
    d2[(j+x)%SIZE] = sum3;
    d2[(j+y)%SIZE] = sum4;
    predecessor()->b = b;
} /* for j */
}

void D::Init(void)
{
    int j;
    for(j=0;j<SIZE;j++) {
        int x,y;
        x = ROW%(SIZE/2);
        y = ROW%(SIZE/2)+SIZE/2;
        a1[j] = 1.0 + SIZE * x + j;
        a2[j] = 1.0 + SIZE * y + j;
        b[j] = 1.0 / (1.0 + SIZE * j + x);
        b[j+SIZE] = 1.0 / (1.0 + SIZE * j + y);
    }
}

main()
{
    D::Init();
    start_timer();
    psections {

        ON PP/2;
        D::matrix_mult1();

        ON PP/2;
        D::matrix_mult2();

    }
    stop_timer();
} /* main */

```


APPENDIX B

Source Code for the Banded System Solver

Most of the source code for the banded system solver is given in this appendix. Set-up and bookkeeping code has been removed. Gaps in the provided code are represented with "...". Phase 0 is initial setup. Phase 1 corresponds to Equation 7; Phase 2 corresponds to Equation 8; Phase 3 corresponds to Equation 9 in Chapter 7.

```

#define N 1008
#define P 4
#define B 33
#define NUM_ITERATIONS 200
/* CONC and PP are defined on the compiler line with -D */

typedef double TYPE;
typedef TYPE * VECTOR;
typedef TYPE ** MATRIX;

#define NC (P-1) /* number of cuts */
#define BA ((B-1)/2) /* width of cut */
#define RI (NC*BA) /* Rows in interface part */
#define CI RI /* Cols in interface part */
#define RD N-RI) /* Rows in domain part */
#define CD N-CI) /* Cols in domain part */
#define PS (N-RI)/P) /* partition size */

#define ID1 (this - VP1)
domain DOMAIN1 {
    TYPE templ[CONC];
    int index[CONC];
    TYPE poly_x_o[CONC/P];
} VP1[RI];

#define ID2 (this - VP2)
domain DOMAIN2 {
    TYPE temp[CONC];
    int index[CONC];
    TYPE x_i[CONC][P]; /* you don't really need this many */
} VP2[PS];

TYPE mono_temp[CONC];
TYPE mono_templ[CONC];
TYPE x_o[RI];
TYPE OutVec_0[RI];
TYPE OutVec_1[RI];

```

```

TYPE OutVec_2[RI];
TYPE OutVec_3[RI];

main(int argc, char *argv[])
{
    int iters;
    MATRIX A;
    VECTOR b;
    VECTOR x;
    MATRIX A_ii_inv[P];
    TYPE r_o[RI];
    MATRIX A1[P];
    MATRIX A2;

    /* parse the command line */
    CommandLine(argc,argv);

    /* Create a banded input system (diagonally dominant)*/
    A = AllocMatrix(N,N);
    x = AllocVector(N);
    b = CreateProblem(A, x);

    /* rearrange some rows of the A matrix to get the right shape */
    {
        ...
    }

    /*
    PHASE 0: This is all of the stuff that gets done ONCE
              and only ONCE, for all solution iterations.
    - pre-compute the A_ii_inv
    - pre-compute A1 <- A_0i * A_ii_inv (P of them)
    - pre-compute A2 <- Inverse(A_00 - (A_0I*A_II_inv*A_I0))
    */
    {
        ...
    }

    StartTimer();
    for(iters=0; iters<NUM_ITERATIONS; iters++) {
        int i,j,k;
        TYPE sum[RI];
        VECTOR b_i;
        MATRIX A_i0;
        int entry;

    #undef STATEMENT
    #define STATEMENT \
        ON PP/CONC; \
        OUT OV; \
        { \
            VECTOR b_i; \
            b_i = b + (SECTION*PS+RI); \

```

```

[domain DOMAIN1]. {
  for(i=0;i<CONC;i++) {
    temp1[i] = (TYPE)0.0;
    index[i] = (ID1%(RI/CONC))+((RI/CONC)*i);
  }
  for(k=0;k<PS;k++) {
    for(i=0;i<CONC;i++) {
      temp1[i] += A1[SECTION][index[i]][k] * b_i[k];\
    }
  }
  k = (RI/CONC) * (SECTION % CONC);
  for(;k<((RI/CONC)*((SECTION%CONC)+1));k++) {
    mono_temp1 = VP1[k].temp1;
    for(i=0;i<CONC;i++) {
      entry = (k%(RI/CONC))+((RI/CONC)*i);
      OV[entry] = mono_temp1[i];
    }
  }
}
}

psections {
# undef SECTION
# define SECTION 0
# undef OV
# define OV OutVec_0
STATEMENT
# undef SECTION
# define SECTION 1
# undef OV
# define OV OutVec_1
STATEMENT
# undef SECTION
# define SECTION 2
# undef OV
# define OV OutVec_2
STATEMENT
# undef SECTION
# define SECTION 3
# undef OV
# define OV OutVec_3
STATEMENT

ON 1;
IN OutVec_0,OutVec_1,OutVec_2,OutVec_3;
OUT r_o;
{
  for(j=0;j<RI;j++)
    r_o[j]=b[j]-(OutVec_0[j]+OutVec_1[j]+OutVec_2[j]+OutVec_3[j]);
}

ON P * PP / CONC;
IN r_o;

```

```

OUT x_o;
{
  int conc;
  /* phase 2 */
  if ( (P*PP/CONC) < PP )
    conc = CONC/P;
  else
    conc = 1;
  [domain DOMAIN1]. {
    for(i=0;i<conc;i++)
      poly_x_o[i] = (TYPE)0.0;
    for(k=0; k<RI; k++) {
      for(i=0;i<conc;i++) {
        poly_x_o[i] += A2[ID1+(i*RI/conc)][k] * r_o[k];
      }
    }
    /* reduce poly_x_o to x_o (mono) */
    for(i=0;i<RI/conc;i++) {
      for(j=0;j<conc;j++) {
        x_o[i+(j*RI/conc)] = VP1[i].poly_x_o[j];
      }
    }
  }
}

#undef STATEMENT
#define STATEMENT \
ON PP/CONC; \
IN x_o; \
{ \
  int off = (SECTION*PS+RI); \
  A_i0 = A + off; \
  b_i = b + off; \
  [domain DOMAIN2]. { \
    int col; \
    for(i=0;i<CONC;i++) { \
      temp[i] = (TYPE)0.0; \
      index[i] = (ID2*(PS/CONC))+((PS/CONC)*i); \
    } \
    for(k=0;k<CI;k++) \
      for(i=0;i<CONC;i++) \
        temp[i] += A_i0[index[i]][k] * x_o[k]; \
    for(i=0;i<CONC;i++) { \
      temp[i] = b_i[index[i]] - temp[i]; \
      x_i[i][SECTION] = (TYPE)0.0; \
    } \
    k = (PS/CONC) * (SECTION % CONC); \
    for(;k<((PS/CONC)*((SECTION%CONC)+1));k++) { \
      mono_temp = VP2[k].temp; \
      for(i=0;i<CONC;i++) { \
        col = (k*(PS/CONC))+((PS/CONC)*i); \
        for(j=0;j<CONC;j++) { \
          x_i[j][SECTION] +=

```

```
        A_ii_inv[SECTION][index[j]][col]*mono_temp[i];\
    }
}
}
}

# undef SECTION
# define SECTION 3
STATEMENT
# undef SECTION
# define SECTION 2
STATEMENT
# undef SECTION
# define SECTION 1
STATEMENT
# undef SECTION
# define SECTION 0
STATEMENT

} /* end of psections statement */
} /* iters */
StopTimer();
Time=ResetTimer();
...
}
```