Functional Programming And Fine-Grain Multithreading For High-Performance Parallel Computing

Chinhyun Kim

CENG Technical Report 94-16

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4484

May 1994

FUNCTIONAL PROGRAMMING AND FINE-GRAIN MULTITHREADING FOR HIGH-PERFORMANCE PARALLEL COMPUTING

by

Chinhyun Kim

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

May 1994

Copyright 1994 Chinhyun Kim

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

Chinhyun Kim

under the direction of his Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of

DOCTOR OF PHILOSOPHY

Barbara Solomon

Dean of Graduate Studies

Date April 13, 1994

DISSERTATION COMMITTEE

Chairperson

Dedication

To my wife Inja and my daughter Ellen. Without your love and understanding, I would never have been able to finish the program.

Acknowledgements

I am forever indebted to Professor Jean-Luc Gaudiot who has been my advisor and a friend. His confidence in me even at times when I doubted myself has helped me finish this long journey. I would also like to express my gratitude to professors Alvin Despain and Ellis Horowitz for serving on my dissertation committee. Their inquisitive questions have stimulated my research. I thank professors Sandeep Gupta and Victor Prasanna for being on my Ph.D guidance committee. I am grateful to Professor Wlodek Proskurowski of the mathematics department who have shown great interest in my work.

I thank my brother and sisters' families for all the moral support they have given me. Not once have they asked that universally hated question "When do you finish?" I especially thank my sister and brother-in-law in Saratoga who have invited our family to many fun-filled trips which I would not have been able to afford otherwise. My special gratitude goes to my parents-in-law who have supported my family in many many ways during my study at USC. At last, but not least, I thank my parents and grandparents for raising and educating me. Without them, I would not be here.

I would like to thank my former group members Professor Andrew Sohn and Dr. Chih-Ming Lin for giving me advice and encouragement. I thank my colleagues Namhoon Yoo, Daekyun Yoon, and Moez Ayed who have become my good friends over the years. Numerous discussions or should I say arguments I had with Namhoon have stimulated my research greatly. I am always impressed by Daekyun's expertise in the Unix system. He has always managed to have the best and the latest software available. Moez got me started in going to the gym regularly. I thank him for keeping me fit. Hiecheol Kim has replaced me as the first person to arrive at the office. I thank him for showing me diligence. I also acknowledge group members Yung Chen, Steve Jenks, Wen-yen Lin, Susan Mabry,

Chulho Shin, and Hung-Yu Tseng. Special thanks goes to Mary Zittercob, Rohini Montenegro, and Joanna Wingert for their assistance.

Contents

List Of Tables

List Of Figures

Abstract

When imperative languages are used to transform algorithms to programs, artificial control dependencies are inevitably introduced. This is due to the sequential computation model on which the imperative languages are based. Consequently, the parallelism that existed in the original algorithm is "corrupted" by the transformation process making it difficult for a compiler to recover. This is the main cause for the three challenges (*programmability*, *performance*, and *portability*) facing the parallel computing community today.

This thesis is based on the belief that functional programming may provide the most natural path toward general-purpose parallel computing. In this regard, the thesis is logically divided into two parts. The first part of the thesis is concerned with demonstrating the viability of functional programming. Using the SISAL functional language, we show that functional programming can indeed satisfy the three challenges stated above.

The target architecture of the current SISAL compiler is a shared-memory parallel architecture. The drawback of such an architecture is limited scalability. For example, commercial parallel shared-memory machines available today are typically limited to less than thirty processors. The second part of the thesis, therefore, is concerned with the development of a basic execution model for a next generation compiler whose target machines will have distributed-memory parallel architectures.

To address the latency problem which is the key performance issue in such architectures, we adopt a fine-grain multithreaded execution model whose main objective is to *tolerate* rather than to *reduce* latency. Our main emphasis in the development of a basic execution model is on issues related to loop level parallelism and array handling since our main interest is in scientific/numeric application domain. We identify the weaknesses in the conventional activation mechanism

and array handling techniques and develop more efficient techniques. We describe a new dynamic loop activation scheme which is more efficient and propose a new array handling method which is cheaper to use and better utilizes the network resources. Finally, we propose a run-time parallelism detection and exploitation technique in the framework of dynamic data-flow model which is useful in many scientific applications.

Chapter 1

Introduction

The current state of parallel computing bears a strong resemblance to that of the early days of serial computing when most programs were written in assembly languages. This chapter describes the research objectives which are motivated by the belief that functional programming may provide the most natural path toward efficient, general-purpose parallel computing.

1.1 Motivation

At the risk of being too simplistic, one can argue that a major obstacle that is blocking the path toward parallel computing today is the difficulty of writing good parallel programs within some reasonable time. This difficulty is attributed to the manual parallelization process which usually entail parallel programming. This process should not have to be an inherent component of parallel programming. Nevertheless, it exists because the automatic parallelizing compiler usually cannot extract sufficient amount of parallelism from a given program. The culprit is neither lack of parallelism in the algorithm nor a deficiency of the compiler, but the programming language in which the algorithm is represented.

Most parallel programming languages in use today are imperative languages which closely reflect the execution mechanism of the processor on which they execute. In other words, the underlying execution model of imperative languages is that of state machine which has a notion of sequencing. This language semantics force programmers to represent algorithms into a sequence of instructions

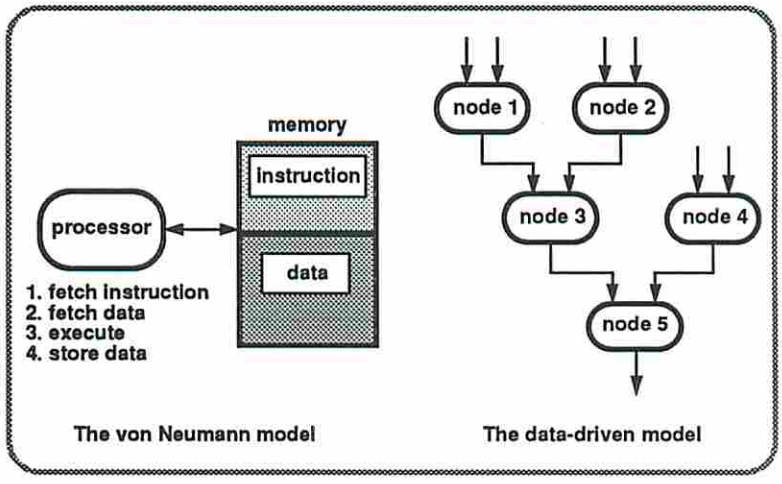

Figure 1.1: In the von Neumann model, a processor sequentially fetches instructions and data from memory for execution. In the data-driven execution model, a program is represented as a graph in which nodes can be viewed as virtual processors. They execute as soon as the input operands become available on the input edges.

describing *how* to compute rather than *what* to compute [10, 58]. This approach inevitably introduces artificial dependencies which are difficult for a compiler to discern from true dependencies [19, 61].

With the current parallel programming approach, it is imperative for a programmer to be familiar with the architecture and the execution mechanism of the target machine. This is because the programmer must be able to choose the most appropriate mechanism available to exploit the type of parallelism at hand. One undesirable side-effect is that, since the program optimization is being done at the application program level, the program becomes less portable as the parallelization process continues.

To overcome the problem of portability, a number of software packages are available. The common approach is to provide a set of standard library routines which can be combined with programs written in conventional C or FORTRAN. Representatives of such software are Linda [2, 22], Parallel Virtual Machine (PVM) [33], Program Composition Notation (PCN) [41], and p4 [18]. The use of these software packages, however, does not free programmers from the parallelization process.

This thesis is motivated by the belief that functional programming may provide the most natural path toward parallel computing. The computation model of functional languages, unlike imperative languages, is not based on state transition [58]. The basic underlying computation model is that of *function* which makes every operation side-effect free, *i.e.*, the output of an operation (function) is only dependent on the input parameters. Furthermore, the executability of a function is only determined by the availability of input parameters. Therefore, when an algorithm is described in a functional language, the parallelism implicit in the algorithm is carried over to the language representation without artificial dependencies creeping in. Hence, a compiler can extract all existing parallelism from a program relatively easily. Based on the strategy which reflects the execution mechanisms of the underlying target machine, the compiler can then generate efficient parallel code.

1.2 Research Objectives

We have general-purpose computing in the serial computing arena. In other words, programmers no longer program in assembly languages to get better performance except for some special instances. Our ultimate goal then, is to provide an environment for general-purpose parallel computing. In the parallel computing arena, general-purpose computing is when programmers no longer have to customize programs to "fit" the underlying machine architecture in order to get good performance.

With this ultimate goal of providing general-purpose parallel computing in mind, we have two research objectives. The first objective is to demonstrate the viability of functional programming in achieving this goal. The methodology chosen to address this objective is by examining the programmability and performance issues together. To this end, an experiment will be performed which will allow us to compare the functional and conventional approaches in terms of these two issues. We will be using the SISAL functional language and the Optimizing SISAL Compiler (OSC) which has shown that SISAL programs compiled by OSC could be competitive with FORTRAN programs [21, 78].

The second objective is to develop a basic execution model for a next generation SISAL compiler whose target machines will have distributed-memory parallel architectures. Distributed-memory parallel architectures are becoming popular as they are currently the most economical means to provide performance scalability [12, 54, 57, 86]. Our approach will be based on a fine-grain multithreaded execution model because we believe that, for parallel architectures, latency tolerant approach is better than latency reduction approach [68].

As our area of interest is scientific/numeric application domain, the emphasis will be given to efficient handling of loops and arrays. Specifically, the conventional loop activation technique can be inefficient in terms of processor utilization resulting in lower performance. The inefficiency arises from the activation mechanism and context switches that occur when multiple loop instances are activated on a same processor. Our objective is to develop a new loop activation technique which will better utilize processors. In terms of array handling, I-Structures is the most well-known scheme. This method, however, is expensive in terms of required hardware support and can cause longer latency through over-utilization of the network. We will develop alternate array handling scheme which is cheaper to use and better utilizes the network resources.

1.3 Outline of the Dissertation

This thesis consists of seven chapters. Chapter 1 includes motivation and research objectives. Chapter 2 is a survey of previous and ongoing research in architectures for exploiting fine-grain parallelism and compiler technologies for efficient execution of functional languages. This chapter shows that while the basic execution model has not changed much, the architecture has changed greatly. The chapter also reports developments in compiler technology which makes it possible to deliver performance through functional programming.

Chapter 3 reports on the viability of functional programming approach for general-purpose parallel computing. The chapter is based on an experiment which is aimed at addressing the issues of *programmability* and *performance* together in the context of parallel computing. The discussion concentrates on three different

implementations of a multigrid elliptic partial differential equation solver. The conclusion of the experiment is that functional programming can indeed deliver performance as well as programmability.

Chapter 4 reports on the development of a new activation management technique called the Multiple Iterations per Activation (MIpA) which utilizes processors more efficiently than the conventional technique. The basic idea of the scheme is to group multiple activations into a single activation such that more threads are available within the activation to overlap communication latency. This means that there is less probability of switching to a different activation which incurs overhead. Simulation using a number of benchmark programs shows that the MIpA model clearly performs better than the conventional method.

Chapter 5 reports on an optimized array handling scheme called the Direct Array Injection Technique (DAIT) which works in conjunction with the MIpA technique. This method directly forwards array elements from the producer to the consumer(s) without having to store the array in a global heap space. The advantages of this method are that dynamic memory allocations/deallocation is not needed and the number of remote operations is reduced. This array handling scheme is applicable in those cases where the life-time of the array producer and consumer(s) overlap. In other words, DAIT is applicable in those situations where arrays behave like temporary variables.

Chapter 6 reports on the work performed at extraction and exploitation of run-time parallelism in the framework of dynamic data-flow execution model. The motivation of this work is that in many scientific applications, though there may be significant amount of parallelism within a loop, it is very difficult and in many cases impossible for a compiler to extract parallelism due to the way the array subscripts are represented. Based on the Token Relabeling method, a technique is developed which can efficiently extract and exploit parallelism at run-time. The method is verified through simulation.

Chapter 7 summarizes the main contributions of research reported in the thesis and suggests further research issues.

Chapter 2

Background

This chapter describes past and ongoing research aimed at exploiting fine-grain parallelism. The first section discusses the development in the architecture arena where there has been an evolution from pure data-flow architectures to hybrid architectures which feature both the data-driven and the von Neumann execution models. The motivation has been to provide efficient sequential code execution while providing mechanisms to exploit fine-grain parallelism. Section two discusses the advances in compiler technology aimed at efficient execution of functional languages on conventional parallel machines. Most of the information contained in this chapter is based on an earlier survey work [45].

2.1 The Architecture Evolution

We now describe past and present work towards achieving high-performance parallel computing based on the data-driven strategy. We will see that although the basic principle of the computation model has not changed, the machine architecture has gone through major changes. Starting from an architecture that is radically different from the conventional von Neumann architecture, it has gradually evolved to include many features of the von Neumann processor.

2.1.1 Micro Data-Flow : The First Generation

The first generation architectures were characterized by an enthusiastic direct implementation of data-flow principles at the architecture level. This means that synchronization and partitioning were all done at the architecture level.

2.1.1.1 Some Micro Data-flow Architectures

This section discusses some of the first generation data-flow machines which include the Manchester Machine [50, 90, 94] and SIGMA-1 [55, 56]. The discussion of the MIT Monsoon dataflow machine [26, 53] is also included although its architecture is quite different from the others in terms of token matching mechanism. In a sense, Monsoon is the last, in chronological terms, of the pure data-flow architectures. Note that due to space limitations, static data-flow architectures are not included in this survey. Interested readers are referred to [90]. More recently, an interesting work on a static data-flow architecture has been performed by [91]. The seminal work of Dennis [30, 31] in the development of a static data-flow architecture which stimulated the development of the dynamic data-flow architectures should be further noted.

The common objective was to efficiently exploit a large amount of fine grain parallelism extracted by the compiler. Although actual implementation differs from one machine to another, they had the following common characteristics :

- Data elements are physically transported in the form of scalar tokens within a processing element or across different processing elements.
- The firing rule as defined in Section 1 is implemented as one of the basic functions provided by the hardware.
- An instruction is scheduled dynamically when its input data tokens arriving at a processing node are properly synchronized (matched).

Manchester Dataflow Computer: The Manchester dataflow computer is one of the first machines to be actually built. It was developed at the University of Manchester in the United Kingdom. The processor consists of four hardware

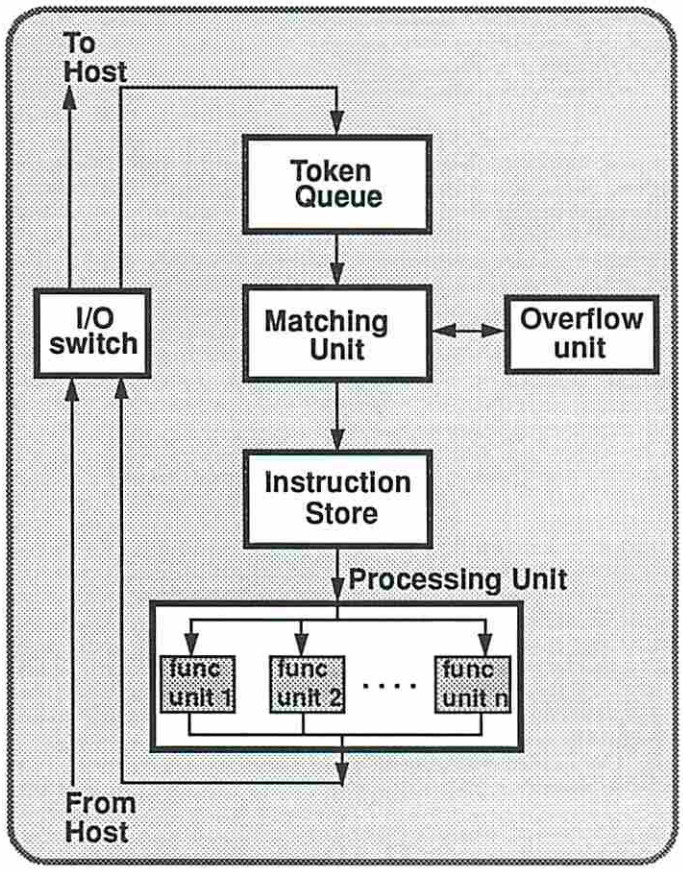

Figure 2.1: Processor organization of the Manchester dataflow machine.

modules which are connected by a pipelined ring (Figure 2.1). The modules operate asynchronously from each other and the data tokens are transported in packets around the ring structure. Although the packets are transferred at a maximum rate of 4.37 million packets per second on the machine, the ring is rated for up to 10 million packets per second. An I/O Switch is provided for the interface between the host and the machine and also for multi-computer implementations.

The *Token Queue Unit* consists of a 32K word circular FIFO store with three buffer registers. The main function of the Token Queue is to store initial data tokens as well as to smooth out any uneveness in the rate of token generation and consumption between different modules. A word is 96 bits long which is the length of a token. The format of a token is shown below. The length of each field, in number of bits, is indicated in parenthesis. The marker field indicates whether the token belongs to a user or a system operation. Other fields are self-explanatory :

Token ≡ [data (37).tag (36).destination (22).marker (1)].

The tokens that are temporarily stored in the Token Queue Unit eventually arrive at the *Matching Unit* which is responsible for instruction scheduling. An instruction is deemed executable when the Matching unit finds all of its input operands available. The input operands of an instruction are found by associatively matching the arriving token with the waiting tokens. The matching is achieved by using the subfields of the token which are 54 bits long. They are the tag field (36 bits) and the instruction address field (18 bits) of the destination field. A 16 bit hashing function is applied and the resulting value is used to address the eight hash tables in parallel. If a match occurs, one of the tables would produce the matching token. The matching token and the incoming token are then formed into a single token and are sent to the *Instruction Store Unit.* If a matching operand is not found, the arriving token is stored in the Matching Unit or in the *Overflow Unit* if all table entries are full. Incidentally, all instructions are either monadic or dyadic operations. The resulting token is 133 bits long and consists of :

Token ≡ [data (37).data (37).tag (36).destination (22).marker (1)].

The Instruction Unit uses the instruction address field of the incoming token to access the instruction. The instruction address field is divided into the segment field (6 bits) and the offset field (12 bits). As in conventional processors, the segment field is used to access a location in the segment table to retrieve a 20 bit segment base address. The offset value is then added to the base address to fetch the instruction that would operate on the input operands. The instruction consists of two fields. One is the opcode field (10 bits) and the other is the destination field. There can be up to two destination fields and one can contain a literal data. Again, the instruction along with the rest of the token fields are formed into a packet and sent to the *Processing Unit* for execution :

Token ≡ [data (37).data (37).opcode (10).tag (36).destination (22).
destination (optional).marker (1)].

The Instruction Unit can contain an array of up to 20 functional units in which each unit is implemented by a microcoded bit-slice processor. The maximum instruction rate for each functional unit is 0.27 MIPS which makes the processor capable of reaching its maximum throughput of about 5 MIPS. The resulting data is formed into a 96 bit token which is then either sent to the host or circulated back to the Token Queue Unit for further processing. This completes the pipeline cycle.

Several conclusions were drawn from the Manchester Project :

1. A wide variety of programs indeed contain sufficient parallelism to fully utilize the functional units.
2. Compiler optimization is required to generate more efficient code.
3. Storing of all data structures in the Matching Unit can create a high overhead.
4. Even without data structures, the Token Queue and the Matching Unit utilization is still high.

ETL SIGMA-1: ETL SIGMA-1 is a data-flow computer developed by the Electro Technical Laboratory (ETL) in Japan. Currently, it is the largest operating data-flow computer based on the tagged token data-flow architecture. The system consists of 128 processing elements and 128 structure store elements connected in a two-level network. A two-stage global network connects 32 group nodes in which each group node consists of four PEs and four SEs connected in a local network. In addition, there are 16 maintenance and a host processor to handle operations such as I/O, system monitoring, performance measurements, etc.

A processing element consists of five hardware units which are divided into two pipeline stages (Figure 2.2). The first pipeline stage (firing stage) consists of

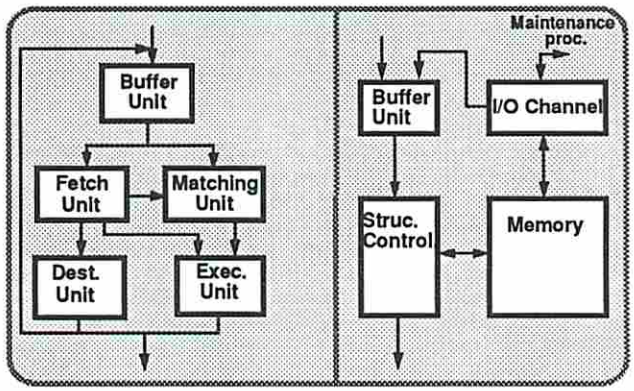

Figure 2.2: Sigma-1 has dedicated hardware structure handlers in addition to the processing elements. The organization of the processing element is shown at left while that of the structure controller is shown at right.

the Buffer Unit, the Matching Unit and the instruction Fetch Unit. The Buffer Unit is a FIFO queue which can store up to 8K tokens where a token is 89 bits in length. The function of this unit is similar to that of the Token Queue Unit of the Manchester machine. When an input operand token of a dyadic operation arrives at the Matching Unit, a search is initiated to find the matching token from the waiting pool of tokens. An associative search is done using the tag field of the token. The token format is as follows :

$$\text{Token} \equiv [\text{pe (8).itag (8).tag (32).c (1).type (8).data (32)}].$$

The pe field indicates the processing element to which the token is to be sent and executed. The itag indicates the type of a token, *e.g.*, a token may be a user data token or a maintenance token. The c field is used to indicate the validity of the token *i.e.*, a token is invalid when the field is set. The type field indicates the data type. The current implementation only allows for a single precision floating point representation of 32 bits. The tag field is further divided into the following fields :

$$\text{Tag} \equiv [\text{i (10).base (8).offset (10).flg (4)}].$$

The i field indicates the loop iteration to which the token belongs. The base and the offset fields are used to fetch instructions. The flg field indicates the type of matching functions. For example, the SIGMA-1 machine provides a "sticky" token matching function for loop invariants. While regular tokens are deleted from the Matching Unit once a match occurs, a sticky token is not deleted. This function enhances performance since a token does not need to be recirculated.

An instruction format is shown below. The length of an instruction can vary from a word to three words in length in which a word is 40 bits long. The fan-out of an instruction is three at the maximum, *i.e.*, there can be at most three destinations. Also, a literal value can be included in an instruction. The ndest field indicates the number of destinations :

```
Instruction ≡ [literal (40).opcode (8).ndest (2).dest0 (20).
               dest1 (20).dest2 (20)].
```

The Execution Unit and the Distribution Unit belong to the second stage (execution stage) of the processor pipeline. The Execution Unit consists of an integer ALU, a floating-point ALU, a multiplier, and a structure address generator. The Distribution Unit is responsible of forming a result into a token and distributing it to its destinations as specified in the instruction. The two stage processor is clocked at 10 MHz. and is estimated to provide 1.7 MFLOPS. The programming language chosen for the SIGMA-1 project is a single assignment language called DFC (Data-Flow C) which is a derivative of C.

MIT Monsoon: The common drawback of the aforementioned data-flow machines is the overly complex and expensive hardware needed to implement the matching function. The key architectural breakthrough of the Monsoon data-flow machine which evolved from the Tagged Token Dataflow Architecture (TTDA) [8] is the Explicit Token Store (ETS) mechanism. Using this mechanism, a matching function becomes a simple read-write operations on a conventional memory device instead of an associative search. The net result is simpler processor hardware and faster token matching.

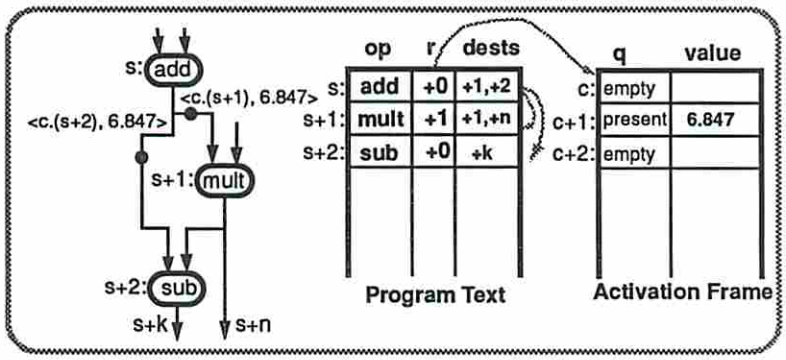

Figure 2.3: In the ETS mechanism, tokens are matched directly by using the offset generated at compile-time.

The ETS mechanism relies on a compile-time analysis which assigns to each token a synchronization point. This information is encoded into the instruction to be used at run-time to directly access the synchronization location. The assigned synchronization point is really an offset from the base of a memory block which is dynamically allocated to a group of instructions called a *code block*. A code block can be a function body or a loop body in the source program. The allocated memory block is called a *frame* memory and it provides the synchronization space to the tokens that belong to the same code block.

In Monsoon, two values `FP` and `IP` constitute a tag. The `FP` is the base address of the frame memory and the `IP` is the instruction pointer. The `FP` uniquely identifies the instance of a code block because every code block activation gets its own unique frame memory block. When a token arrives, a processor can compute the exact synchronization location within the frame memory using these two values :

1. The `IP` is used to fetch the instruction.
2. The offset stored as part of the instruction is fetched and added to the `FP`.
3. The computed frame memory location is accessed and the presence bit is checked.
4. If the presence bit is set, a matching token exists. The matching token is read from the frame memory and the two data values are sent to the execution unit. If the presence bit is not set, the incoming token is stored.

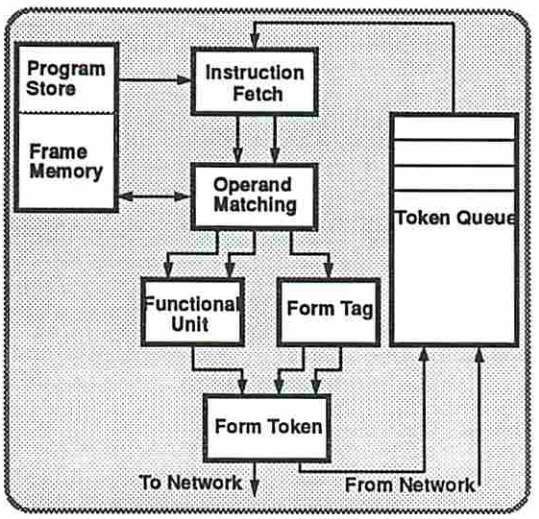

Figure 2.4: Organization of the Monsoon processor.

The Monsoon processor architecture is shown in Figure 2.4. Unlike the previous data-flow machines, the first unit in the pipeline is the Instruction Fetch Unit. This of course is due to the ETS mechanism which requires the offset value to compute the synchronization location. Because of the simpler token matching hardware, the processing time of the match unit is comparable to the other units in the pipeline which results in a more balanced and efficiently executing pipe. For most other data-flow machines which use associative matching, the match unit is the bottleneck which hampers the pipeline throughput.

2.1.1.2 Other Architectures

Other projects are being pursued elsewhere. They include, but are not limited to the following list:

- The RMIT/CSIRO Data-flow architecture.
- The Datarol.
- The Epsilon project.
- The Rapid.

The RMIT/CSIRO Data-flow machine was studied at the Royal Melbourne Institute of Technology and the Commonwealth Scientific Industrial Research Organization in Australia. The architecture of the machine is evolved from the Manchester Data-flow machine. The design objective is to provide an efficient execution environment for both static and dynamic execution styles. Thus, the architecture provides two modes of token matching. In addition to token matching by tag (dynamic model), tokens can be queued and executed in the arriving order (static model). The motivation for such a hybrid architecture is that applications such as DSP are more suited for static execution while other applications may benefit from a dynamic execution style [1].

A data-flow architecture called Datarol is under study at the Kyushu University in Japan. The datarol architecture has some features characteristic of recent hybrid architectures. For example, instead of dispatching data tokens to the destination instructions, datarol provides a *by-reference mechanism* which allows the destination instructions to fetch input operands. This mechanism makes token copying unnecessary when there are multiple destination instructions. In addition, compile-time optimization can be performed to have an internal register as the operand store for improved performance [4].

The Epsilon-2 architecture is developed at the Sandia National Laboratories as an outgrowth of the Epsilon-1 project. However, unlike Epsilon-1, Epsilon-2 implements a dynamic data-flow execution model. The architecture uses direct token matching similar to that of Monsoon. Its salient feature is the *repeat token* mechanism which is also available on the Epsilon-1 architecture. The purpose of this feature is to reduce the overhead associated with data fanout [48, 49].

A data-flow processor chip set is designed and implemented by a collaboration of the Osaka University, the Mitsubishi Electric Corporation, and the Sharp Corporation [67, 77]. The processor called Qv-1 is based on the dynamic data-driven execution model and consists of five chips. The Rapid processor is implemented on a single board using the chip set which can provide up to 20 MFLOPS of computing power. A unique feature of the Qv-1 processor is its asynchronous pipelining scheme. In the Qv-1 chips, data transfer between two pipeline stages

is achieved by the exchange of send and acknowledge signals between the self-timed data-transfer control circuits. In the future version, the five chip set will be integrated into a single chip.

2.1.2 Hybrids : The Second Generation

Several points became clear as a result of experimenting with the data-flow architectures discussed in the previous section.

First, too many functionalities have been given to the architecture resulting in overly complex hardware making them economically infeasible. The match unit became a bottleneck in the circular pipeline of the data-flow architectures. The ETS mechanism used in Monsoon is one solution in the right direction. By relying more on a compile-time analysis, a simpler synchronization mechanism was developed to reduce the hardware costs and matching time.

Second, dynamic scheduling of every instruction is expensive and in many cases unnecessary. By analyzing the data dependency graphs at compile-time, dependent instructions can be statically scheduled without losing much parallelism. By doing so, many von Neumann processor optimization techniques may be applied. Figure 2.5 shows a portion of a scientific application represented as an IF1 graph [88]. Assuming that the availability of the data tokens representing values `i` and `h` cannot be determined at compile-time, the only instructions that require dynamic scheduling are instructions 1 and 4. The instructions 2,3, and 5 can be scheduled statically according to the data dependency relationship.

This section discusses the recent research activities aimed at addressing these points. The common approach taken by many researchers is a more balanced distribution of functionalities between the compiler and the hardware. It is driven by the fact that, at least in today's technology, pure data-flow architectures are not economically feasible. The resulting solution is also known as *multithreading.*

2.1.2.1 Some Hybrid Architectures

After extensive experimentations with the pure data-flow architectures, the second generation data-flow architectures have gradually become hybrids in an attempt

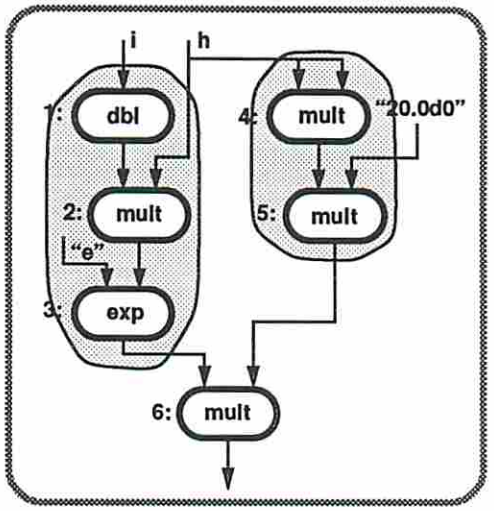

Figure 2.5: A portion of a scientific application is shown as an IF1 graph.

to combine the best features of the conventional von Neumann architectures and the pure data-flow architectures. With such architectures, a sequential stream of instructions can be efficiently executed according to the von Neumann model while the fine-grain parallelism is exploited according to the data-driven model.

P-RISC: The objective of the P-RISC (Parallel-RISC) was to propose an architecture suitable for multiprocessing by starting out with a von Neumann architecture and extending it to provide necessary features to exploit fine-grain parallelism by borrowing from the data-flow architecture principles [74]. The resulting architecture has four new instructions in addition to its conventional RISC instruction set. They are :

1. `fork IPt`
2. `join x`
3. `start v c d`
4. `loadc a x IPr`

In a pure data-flow architecture, token matching is an implicit operation which is transparent to the compiler. Transmission of the output tokens to their respective destinations is also implicit to the compiler. In P-RISC, however, these

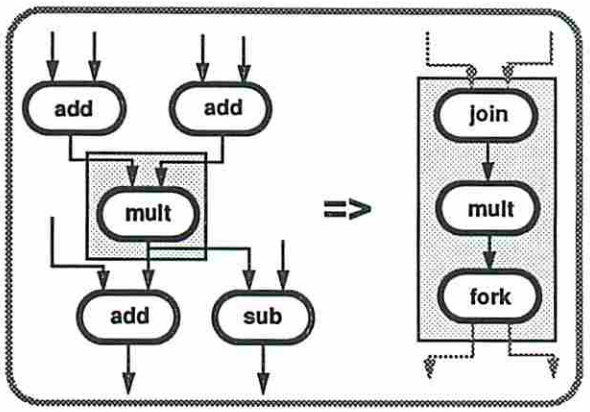

Figure 2.6: In P-RISC, token matching and forwarding is explicit under the control of the compiler.

functions are under the explicit control of the compiler and the first two instructions `fork` and `join` are the direct consequences of this change. The `fork IPt` instruction creates two active threads by queueing two *continuations* `<FP.IP+1>` and `<FP.IPt>` in the token queue. The `join x` instruction toggles the value at frame location `FP+x` which is the storage location of the presence bits. If the result is one, nothing happens. It the result is zero, a continuation `<FP.IP+1>` is inserted in the token queue. Figure 2.6 shows an example in which these two instructions are used to effect data-driven execution style.

The `start v c d` instruction is used to activate a thread belonging to a different activation which may or may not be executing on the same processor. This is accomplished by sending a message of the form `<START, [FP+v], [FP+c], [FP+d]>`. The second message operand `[FP+c]` is the descriptor of the thread to be activated. Upon arriving at the destination, the value at the first message operand is stored at the frame memory location whose offset is indicated by the third message operand and the thread descriptor is inserted into the token queue. This instruction can be used for a procedure call linkage mechanism, *i.e.*, a return value can be sent as the first operand while the second operand indicates the thread that is activated upon receiving a return value.

The `start v c d` instruction is also used to implement *split-phase* remote read operations. This operation is used in data-driven execution to efficiently utilize a processor when a long latency inducing operation is activated. Instead

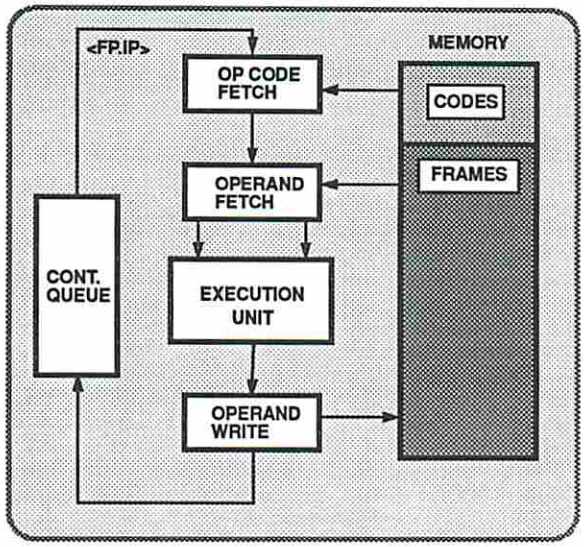

Figure 2.7: Organization of the P-RISC processor.

of idling, a P-RISC processor executes an active thread from the token queue after dispatching a read request message of the form `<READ,a,FP.IP+1,x>`. In the message, `a` is the address of the requested memory location, `x` is the offset of the frame memory location which the returning value is to be stored. `FP.IP+1` is the thread descriptor which is to be activated upon arrival of the requested data. When a `READ` message is received, the memory handler saves the thread descriptor and the offset. After reading the value from address `a`, it executes a `start` instruction which sends the value along with the thread descriptor and the offset value.

To sustain concurrency, a `load` instruction is usually surrounded by instructions such as `jmp` and `fork`. The `loadc a x IPr` is an extension of the `load` instruction which can make coding simpler in many cases. The `loadc a x IPr` instruction generates a `READ` message with `IPr` as the return address and continues execution from `IP+1`. Thus, the `loadc` instruction performs an implicit fork after generating a `READ` message [74].

Work is in progress to implement a processor based on the P-RISC concept. The processor is designated *T (pronounced "start") and is to be extended from the existing Motorola 88110 superscalar RISC processor [11, 32, 76]. The *T processor has a functional unit called the Message and Synchronization Unit (MSU)

in addition to the existing functional units. The two main functions of the MSU is the handling of messages and scheduling of threads. The MSU provides a close coupled register-set model of network messaging by providing a set of transmit and receive registers. User level instructions are available to send and receive messages without the operating system support. In *T, the maximum message length is fixed at 24 words.

In addition to network support, the *T processor provides mechanisms for multithreading. In the MSU, a 64 entry Microthread Stack is provided to store thread descriptors. Scheduled threads are stored in the 8 Scheduled Microthread Descriptor Registers. The threads are scheduled via fixed priority scheme of the Scheduled Microthread Descriptor Registers, the network receiver, and the Microthread Stack.

USC Decoupled Architecture Model: Two factors have influenced the development of the USC Decoupled Architecture Model. First is the need to develop an efficient architecture model for data dependency graphs with variable resolution actors [36]. It has been shown that appropriate fusing of simple nodes into larger macro nodes can improve overall performance without significant loss of parallelism [44]. In such an execution model, however, there can be a great difference in the execution time according to the granularity of the node. As a result, a large buffer is required between the graph stages (match and token formating/routing) and the computation stages (fetch and execute) of the processor [36]. Second, it would be beneficial to borrow advanced pipeline techniques from von Neumann processors to execute sequential streams of instructions belonging to macro nodes.

Figure 2.9 shows the Decoupled Architecture model which consists of two processing units. They are the *Computation Engine* (CE) and the *Data-Flow Graph Engine* (DFGE). The CE is responsible for executing the instructions belonging to a node once the node is scheduled for execution. A high performance conventional von Neumann processor is appropriate for the Computation Engine since the instructions within the node are executed in sequence. The role of the DFGE is that of the scheduler. It performs token matching and once all input tokens of a node is available, the node is scheduled for execution by the CE. The two

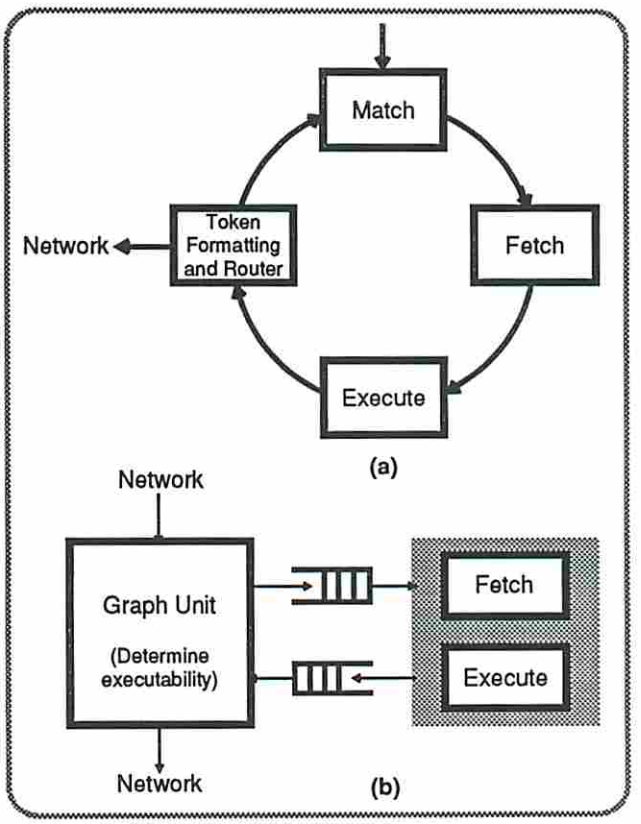

Figure 2.8: Basic data-flow architecture organized as a (a) cyclic pipeline and (b) Decoupled Graph/Computation configuration. Courtesy of Prentice Hall, *Advanced Topics in Data-flow Computing*, J-L. Gaudiot and L. Bic eds., 1991.

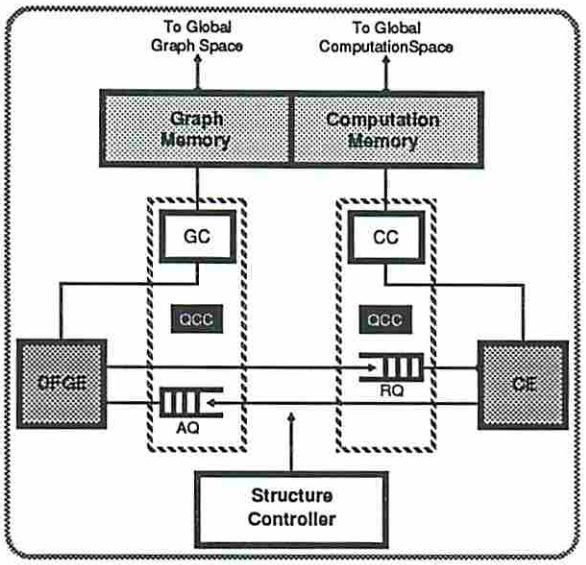

Figure 2.9: Data-flow processor with decoupled graph and computation units. Courtesy of Prentice Hall, *Advanced Topics in Data-flow Computing*, J-L. Gaudiot and L. Bic eds., 1991.

processing units are connected by two queues. The *Ready Queue* (RQ) holds the descriptor of the ready nodes inserted by the DFGE. The *Acknowledge Queue* (AQ) holds the descriptor of the node that has been executed by the CE. Upon receiving the descriptor of the executed node, the DFGE can schedule other nodes which depend on the results of the node.

EM-4: The EM-4 multiprocessor machine was developed at the ETL of Japan. Currently, a machine containing 80 processors is in operation. Similar to other recent data-driven architectures, the EM-4 architecture is based on a multithreading model called the *strongly connected arc model.* In this model, a thread is called a *strongly connected block* (SCB). Once an SCB is initiated, execution continues without preemption until the SCB terminates. Instructions within the SCB are executed sequentially in a control-driven fashion [82].

As in Monsoon, a direct token matching scheme is used in EM-4 [84]. However, the actual mechanism is different from that of the ETS used in Monsoon. There are two main differences from the ETS scheme. First, in EM-4, the offset value which specifies the storage/synchronization location of an instruction's operand

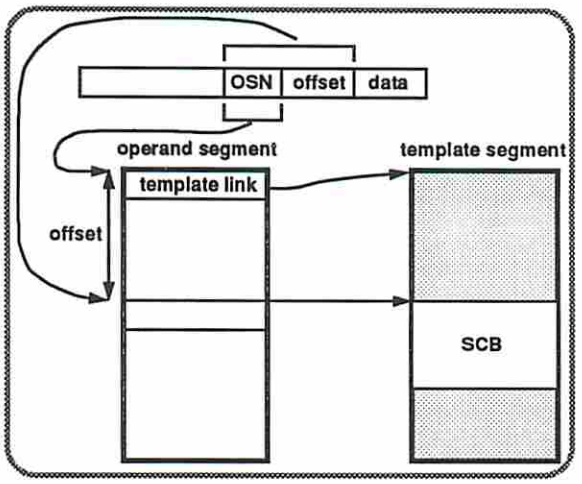

Figure 2.10: The direct matching scheme of EM-4. The operand segment refers to the frame memory and the template segment refers to the code block. The operand segment number OPN and the offset are carried in the data token packet.

is carried in a data token. In the ETS mechanism, the offset value is stored as part of the instruction. Second, the offset not only indicates the synchronization point of tokens, but also acts as the offset for the instruction which operates on the tokens. In ETS, the offset value is only valid for accessing operands within the frame memory. Figure 2.10 describes the direct matching scheme of EM-4.

The heart of the EM-4 machine is the EMC-R single chip processor. The EMC-R processor architecture provides both the data-flow and the von Neumann execution mechanism within the same processor. The processor contains four pipeline stages in which the first two stages are used for token matching functions while the latter two stages are used for the fetching and the execution of the instructions. For the instructions that belong to a thread (or SCB), only the latter two pipeline stages are used once the thread is activated. The Operand Match Unit, the Instruction Fetch Unit, the Destination Unit and the Execution Unit shown in Figure 2.11 correspond to the four stages.

The EMC-R processor chip also contains the Switching Unit (SU) which performs a three-by-three packet switching function. The tokens that are destined for the local processor are first stored in the Input Buffer Unit (IBU) which is a 32 word FIFO buffer. An 8K word secondary buffer is located in an off-chip memory in case of an overflow. Although the EMC-R processor does not provide

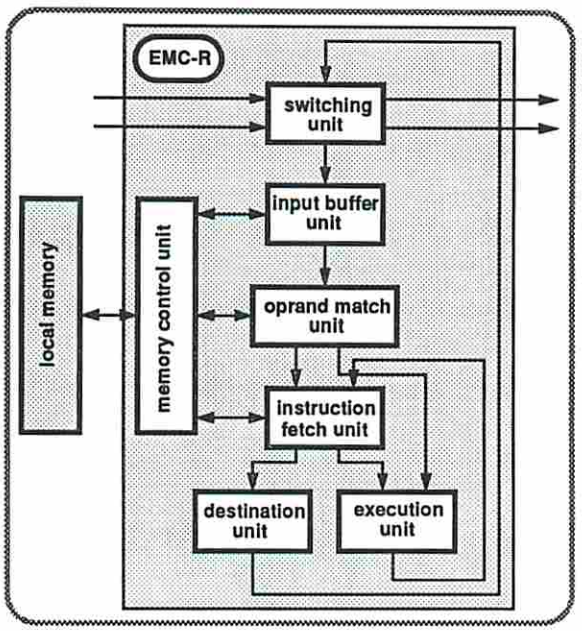

Figure 2.11: The organization of the EMC-R single chip processor.

an on-chip floating-point execution unit, it is possible to interface a commercially available floating-point co-processor chip.

2.1.2.2 Other Architectures

The McGill Dataflow Architecture Model (MDFA) is yet another hybrid architecture model developed at McGill University in Canada. MDFA has a decoupled architecture similar to the USC Decoupled Architecture. In MDFA, a processor is divided into the Instruction Processing Unit (IPU) and the Instruction Scheduling Unit (ISU). The two units interact with each other by exchanging *fire* and *done* signals. A fire signal is sent from the ISU to the IPU which consists of the instruction pointer and the base address of the frame memory (continuation). A done signal is sent from the IPU to the ISU which consists of the continuation of the executed instruction [42].

So far, we have discussed hybrid architectures which evolved from the pure data-flow architectures. There are other hybrid architectures which started from the von Neumann architectures. They will not be discussed in this paper, but they

include the Hybrid Architecture [59, 60], the Message-Driven Processor (MDP) [29], and the Tera machine [3].

2.1.3 Summary

The micro data-flow architectures did not produce overwhelming results in favor of the data-flow approach. On the other hand, they yielded some valuable insights. First, compiler optimization is an important (perhaps major) area. Second, pure data-flow architectures may not be the ultimate answer. Although the data-flow architectures satisfactorily exploit parallelism, their performance in executing sequential code was below standards. This was to be expected since data-flow architectures have no mechanism to exploit locality which is the key to efficient sequential code execution.

The driving force behind the second generation architectures were efficient sequential execution and simpler and faster synchronization mechanism. The result is a hybrid architecture which provides the von Neumann execution model in addition to the data-driven execution model. With the exception of EM-4, these architectures decouple computing tasks from the synchronization tasks. In *T, this is handled by the Message and Synchronization Unit while in the USC Decoupled Architecture Model, the Data-Flow Graph Engine is responsible for the job.

Compilers play an important role since the hybrid architectures depend on compilation strategies much more than the first generation data-flow architectures. Performance can vary greatly depending on the criteria used for thread partitioning. Due to the relatively recent introduction of the hybrid architectures, however, not much work has been reported in the domain of compilation strategies. Figure 2.12 shows how the architectures for the data-driven execution model have evolved.

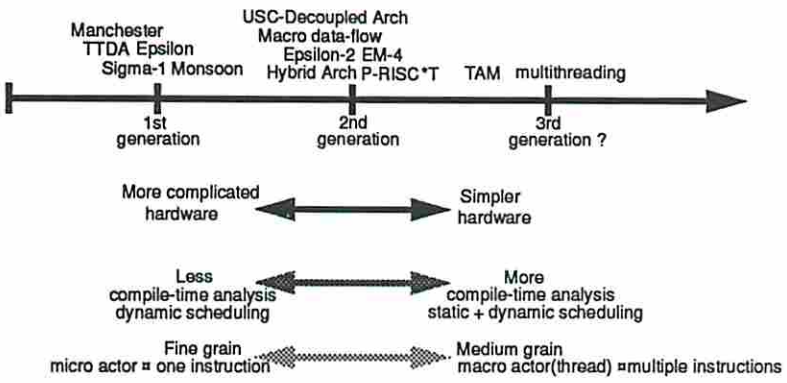

Figure 2.12: The diagram displays how the architectures for the data-driven execution model have evolved from pure data-flow architectures to hybrid architectures.

2.2 Compiler Technology

As it turns out, the principles of functional programs do not necessarily require hardware support. Implicit parallelism through functional programs can indeed be exploited on conventional architectures. Two projects are discussed.

2.2.1 SISAL Compiler

This section discusses the compilation strategy and the performance of the Optimizing SISAL Compiler (OSC) [20]. Currently, OSC is available on many shared memory parallel machines from companies such as Sequent, Silicon Graphics, CRAY, etc. Incidentally, OSC can also be used in uniprocessor workstations running Unix operating system. With this compiler, a SISAL program originally written to run on a small workstation can be recompiled to run on a parallel supercomputer such as CRAY C-90 without program modification.

Traditionally, the main drawback of functional languages (such as SISAL) has been the performance. Although these languages give programmers a high level of abstraction, the lack of performance made them unattractive [70]. However, OSC has shown that high performance parallel computing is possible with functional programming. Experimenting with a number of numerical application benchmark programs showed that programs compiled by OSC performs competitive to that of a FORTRAN code on a single head CRAY Y-MP supercomputer [21]. However,

Program	Source Lines	Time Steps	Problem Size	Application
KIN16	225	42,000	2×14285	Gel Electrophoresis
RICARD	297	40,000	5×1315	Gel Chromatography
CFFT	517	1	524288	Fourier Transform
BMK11A	1003	200	2560	Particle Transport
LOOPS	1116	4000	100 - 1000	Scientific Kernel
SIMPLE	1527	62	100 x 100	Hydrodynamics
UNSPLIT	1937	40	160 x 80	Hydrodynamics
WEATHER	2712	20	420 km	Weather Model

Table 2.1: The list of benchmark programs used to compare the performance of OSC to FORTRAN on a CRAY Y-MP supercomputer.

CRAY Y-MP/864	Concurrent-Vector					Seconds
	One CPU		Four CPUs		Eight CPUs	
Program	Sisal	Fortran	Sisal	Fortran	Sisal	Fortran
KIN16	74.8	79.9	19.7	20.7	11.6	11.2
RICARD	39.8	38.9	13.8	12.5	8.7	7.4
CFFT	0.33	0.42	0.10	0.15	0.07	0.13
BMK11A	4.8	4.2	3.4	3.0	3.3	2.9
LOOPS	16.1	12.9	12.3	11.4	11.3	11.0
SIMPLE	9.4	9.0	3.1	8.8	2.2	8.8
UNSPLIT	10.1	8.0	2.8	5.0	1.7	4.9
WEATHER	7.3	7.0	2.0	6.7	1.2	6.8
Total	162.6	160.3	57.2	68.3	40.1	53.1

Table 2.2: Performance comparison of SISAL and FORTRAN benchmark programs on a CRAY Y-MP supercomputer.

the real advantage is that programs can be run on multi-head CRAY computers without modification with correspondingly improved performance. Table 2.1 describes the type of benchmark programs used in the experiment. Performance results of the benchmarks on a CRAY Y-MP/864 is reported in [21] (Table 2.2). It demonstrates that the programs written in FORTRAN perform slightly better when one processor is used. However, the programs written in SISAL perform better when multiple processors are used.

The main weakness in SISAL in terms of performance has been handling of the aggregate data types such as arrays. Because of the language semantics which require *referential transparency*, arrays often have to be copied. For example, a simple replacement operation of an array element causes array copying which makes the execution time a function of the array size instead of a constant. The problem becomes much worse if the array replacement operation happens to occur inside a loop. It was observed that a SISAL program which initializes an array of 50,000 elements took approximately one hour on a Sequent Balance while the equivalent FORTRAN version took less than one second to complete [19].

The main strategy of OSC is to minimize array copying via an automatic static analysis of the program graphs. Specifically, the objective is to transform array operations that cause copying to either *build-in-place* or *update-in-place* operations. Array operations that result in dynamically growing arrays may be optimized into build-in-place operations if the final array size does not depend on the run-time behavior. Array replacement operations can be optimized into update-in-place operations at the cost of losing some parallelism. The optimizers IF2MEM and IF2UP as shown in Figure 2.13 are responsible for these transformations. The final output of the array optimizers are program graphs in IF2 [96] format with explicit memory management operations.

Although the actual graph analysis aimed at optimizing various array operations are nontrivial, the basic concept is easy to understand. Two simple examples are used to demonstrate the basic intuition. The example of Figure 2.14 shows a loop which initializes an array using the `array_addh` operation. This operation causes the array to grow dynamically. Initially the array contains only one element and grows by one element at each iteration until N elements are produced.

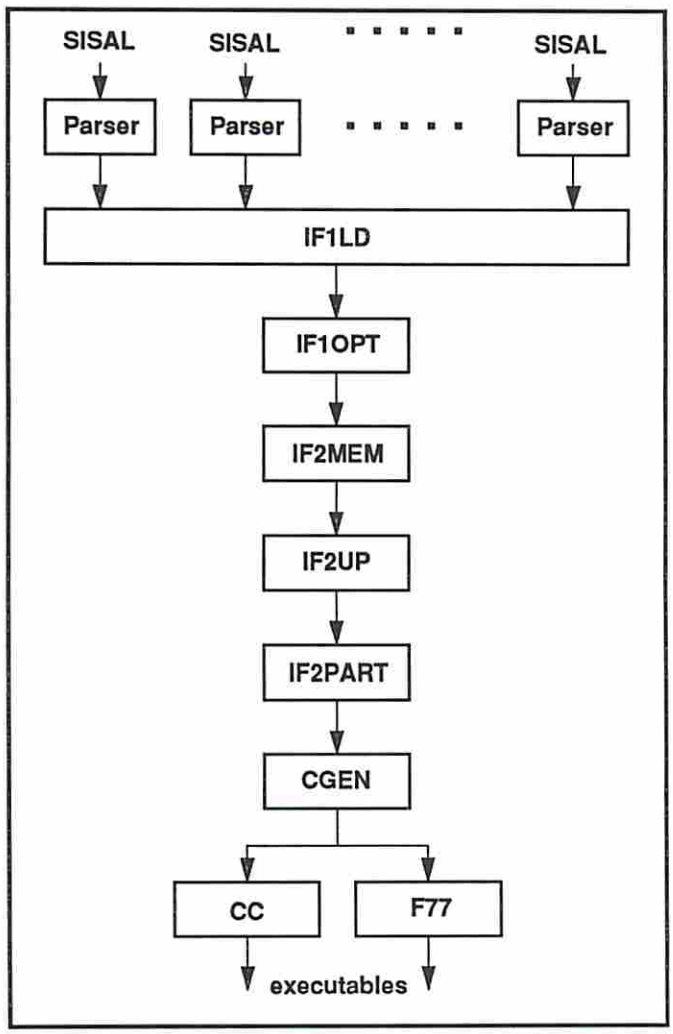

Figure 2.13: The current Optimizing SISAL Compiler goes through a number of optimization phases. IF1OPT performs conventional optimization transformations on the IF1 graphs. The resulting graphs are input to IF2MEM and IF2UP for efficient array usage. After partitioning by IF2PART, C code is generated for final compilation.

```
type OneDim = array [integer];

function Main (N : integer returns OneDim)
      for initial
            I := 1;
            A := array [1:0]
      while (I < N) repeat
            I := old I + 1;
            A := array_addh (old A, 0)
      returns value of A
      end for
end function
```

Figure 2.14: A SISAL program which initializes an array to value zero.

```
for initial
      I := 1
while (I <= 1000) repeat
      I := old I + A[I]
returns array of I
end for
```

Figure 2.15: The IF2MEM optimizer is not able to transform this program segment to preallocate memory because the array size cannot be determined at compile-time.

```
type TwoDim = array [array [real]];

function Main (I,J:integer; A:TwoDim returns TwoDim, real)
      A[I,J : 0.0], A[J,I]
end function
```

Figure 2.16: Without optimization, the two array operations in function Main cause array copying.

A naive implementation would initially allocate one memory cell for the first element. In each subsequent iteration, a memory allocation is performed followed by a copy operation in which the amount of memory allocated and copied is larger by one element from the previous iteration. This process would repeat until all array elements are initialized. This inefficiency can be reduced to one memory allocation and zero copying by preallocating memory for N array elements all at once before entering the loop once the value of N becomes known. Unfortunately, this optimization is not always possible as shown in Figure 2.15 where the build-in-place optimization is not possible because the size of the array A cannot be determined prior to entering the loop.

According to SISAL semantics, the execution of the two array operations as illustrated in Figure 2.16 are implicitly parallel. One is an array replacement operation and the other is an array read operation. Since the relationship between the array subscripts of the two operations is not known, array copying results. In this example, if it can be determined that the array read operation shown is the only array read, the array replacement operation can be transformed into an update-in-place operation by forcing the array read operation to occur first. The IF2UP optimizer looks for such cases during its graph analysis.

In addition to the array optimization analysis performed by the IF2MEM and the IF2UP optimizers, IF1OPT performs well known conventional optimization techniques such as :

- Function inline expansion.
- Loop invariant removal.
- Record fission.

- Common subexpression elimination.
- Loop fusion.
- Constant folding.
- Dead code elimination.

The target architecture of the current OSC is MIMD using a shared memory model. There exist several projects aimed at implementing SISAL on other architectures. One of the earliest such efforts is the development of a SISAL compiler for the Manchester data-flow machine [16]. More recently, porting of OSC to parallel machines based on a distributed memory model has been performed [52]. SISAL has also been implemented on a von Neumann/data-flow hybrid architecture called the Advanced DAtaflow Machine (ADAM) [72].

The performance of OSC has been reported on [21] which compares the execution times of various numerical benchmark programs written in SISAL and FORTRAN on a CRAY machine. A study reported in [35] emphasizes the run-time memory management issues. In relation to language issues, exploiting SISAL stream data types in non-strict computation is studied in [97] and work involving the language specification of the next version of SISAL is in progress [15].

2.2.2 Threaded Abstract Machine

In the first generation pure data-flow architectures, fine-grain parallel execution is provided by special mechanisms that are built into the hardware. For example, matching hardware is provided to dynamically schedule instructions through token matching. Also, a router hardware is provided to automatically route a result data token to its destination. The Threaded Abstract Machine (TAM) [25, 27] takes the opposing approach in that it tries to provide fine-grain parallel computations on conventional parallel machines which provide none of these hardware mechanisms.

To minimize the overhead of frequent transfer of control among a potentially large pool of ready tasks, a fast context switching capability is a prerequisite to providing fine-grain parallel execution. Otherwise, the benefit of parallel execution can easily be wasted by the context switching overhead. Due to the large

states of modern processors, however, fast context switching is difficult to attain. TAM overcomes this problem by addressing two key issues, namely, the scheduling strategy and the communication mechanisms.

The key point of the TAM scheduling strategy is the exploitation of *spatial locality* as much as possible by taking into account the memory hierarchy of the machine. By so doing, the frequency of (expensive) context switching may be reduced. To explain this idea, let us first assume that there are a number of active code blocks in a processor. A code block is said to be active when a frame has been allocated to it and is ready to execute, *i.e.*, at least one of its threads is ready for execution. An active code block is called an *activation.* Although there can be many activations in a processor, only one thread belonging to an activation is executed at a time.

Now, assume that a thread has just terminated and a new thread needs to be scheduled. There are a number of possible scheduling strategies. For example, a round-robin scheduling can be used in which threads belonging to different activations are scheduled one after another. In this scheme, the current processor state needs to be saved each time a new thread is scheduled causing a large overhead. In TAM, ready threads that belong to the current activation have higher priority than other threads. A new activation is scheduled only when no other thread from the current activation can be scheduled. This strategy incurs much less overhead because it takes advantage of the locality as long as possible. In other words, by scheduling threads from the current activation, values stored in processor registers and cache need not be saved at every thread context switch. A large overhead is incurred only when a thread from a different activation must be scheduled.

Communication is another important element of the TAM model. Without efficient communicating mechanisms, fine-grain parallel execution is not possible due to processor idling caused by long latency. In TAM, the communication is tightly coupled to the computation. Thus, the communication chores are handled by the compiler generated *message handler* instead of relying on the operating system services. A message handler is called an *inlet* and its objective is to quickly inject the received data into the currently executing activation threads. By doing

Program	Source Lines	Problem Size	Application
QS	–	–	Quicksort
GAMTEB	750	8192	Neutron Transport
PARAFFINS	300	–	Isomer Enumeration
SIMPLE	1000	128 x 128	Hydrodynamics
SPEECH	–	–	Speech Processing
MMT	130	60 x 60	Matrix Multiplication

Table 2.3: The list of benchmark programs used in the benchmarking of TAM on a 64-node CM-5 machine.

so, activation that is already resident on the processor can continue executing instead of being switched out and brought back again later at a high cost [93].

To verify the TAM concept, an intermediate language representation called TL0 has been developed. The backend of the Id compiler has been modified to produce the multithreaded code in the TL0 format instead of data-flow instructions. The resulting TL0 is then translated to a C code and compiled by the native C compiler for execution. Seven benchmark programs as shown in Table 2.3 are used in the performance measurements. Dynamic scheduling characteristics of the benchmark programs are reported in [25] (Table 2.4). The 64-node CM-5 was used as a target machine. Preliminary results are encouraging in that the performance on the 64-node CM-5 is comparable to a 16-node Monsoon. (Recall that Monsoon is specially designed and built to support fine-grain parallel execution [25].)

The average number of TL0 instructions per thread on the compiled benchmark programs is approximately six which is comparable to the run-length of a basic block in conventional programs [25]. With such short threads, it would be easy to incur a large context switching overhead. However, the third row of Table 2.4 shows that many threads are indeed scheduled (depending on the applications) during a single *quantum* where a quantum is defined as a single residency of an activation on a processor. Through the scheduling strategy and the communication mechanism described above, the TAM model keeps the context switching cost to minimum by restricting the thread switching to those within the current activation as much as possible.

	QS	GAMTEB	PARAFFINS	SIMPLE	SPEECH	MMT
Avg. Thrd lngth	2.6	3.2	3.1	5.3	6.3	17.6
Thrds/Quantum	11.5	13.5	215.5	7.5	16.7	530.0
Quanta/Invoc.	4.1	3.4	2.7	4.8	21.7	3.4

Table 2.4: Dynamic scheduling characteristics of the benchmark programs on a 64 node CM-5 machine.

2.3 Conclusions

Motivated by the desire to exploit large amount of fine-grain parallelism extracted from programs written in functional languages, research in architecture has gone through two distinct evolutionary phases. The first generation data-flow architecture was motivated by the efforts to directly execute data-flow graphs on hardware with minimum compile-time analysis. Consequently, the resulting hardware became quite complex in order to provide various mechanisms for dynamic data-flow execution.

The second generation architecture is characterized by its hybrid nature. Driven by the need to provide cheaper and more efficient execution mechanisms for both sequential and parallel code segments, hybrid architectures possess features from the von Neumann as well as the data-flow architectures. While the hardware has become simpler, more the compile-time analysis has become very important.

On the other hand, there have been pure software approaches toward executing functional programs on conventional parallel machines. Two research efforts discussed in this chapter [19, 27] are distinguished by their main objectives. The objective of TAM was to see how well fine-grain parallelism, based on multithreading, can be exploited on conventional parallel machines. Conclusion from the TAM work is that while the results are impressive considering the non-existing hardware support for fine-grain parallelism, purely software approach cannot compete.

The objective of the Osc work was to directly compete with imperative languages on supercomputers. The main objective of the compile-time optimization was to minimize run-time array copying as much as possible. The exploited parallelism granularity, on the other hand, is coarse at the loop body level. Performance evaluation has demonstrated that functional programming can compete

with imperative languages on conventional parallel machines based on coarse grain parallelism.

It is still not clear how to partition the work required to support fine-grain parallelism among the architecture and the compiler. We have seen, in this chapter, research efforts that started from the opposite sides. A middle ground needs to be found which best partitions the work between hardware and software. The research efforts discussed in this chapter give us a good starting point.

Chapter 3

Functional Programming

This chapter discusses the viability of functional programming in the context of parallel computing by demonstrating that programmability, portability, and performance can be achieved at the same time without sacrificing any one of them. The discussion is based on the results from an experiment which involves implementing a number of numerical applications using different programming languages within some fixed time frame. The performance of the applications written in SISAL is compared to those written in imperative languages on a number of sequential and parallel computers. This chapter is based on the works published and presented in [63, 64].

3.1 Case Study 1: Multigrid Method

Direct method and iterative method are the two well-known approaches to solve systems of linear equations. Direct method is characterized by the known number of steps (iterations) required to find an exact solution. Gaussian Elimination is one example of a direct method. Iterative method, on the other hand, begins with a guess and iterates until "close enough" approximation is reached. Naturally, the number of required steps is not known a priori. Three basic iterative methods are Jacobi, Gauss-Seidel, and Successive Over Relaxation (SOR).

Iterative method is advantageous to direct method when the coefficient matrix is sparse. In iterative method, the sparsity of a matrix is retained throughout the computation while direct method destroys the sparsity. The disadvantage of the

basic iterative methods is that the rate of convergence decreases as the problem size increases. Of the three methods listed, SOR has the best performance. This, however, is based on the assumption that the optimum value of the weighting parameter ω_{opt} is known which is not the case in general [47].

Conceptually, the slowness of the basic iterative methods is attributed to the fact that they act as a low-pass filter with a fixed cutoff frequency. Initially, the convergence rate is high because the high frequency error components are quickly filtered out. However, once the high-frequency error components are filtered out and only the low-frequency (or smooth) error components are left, the convergence rate becomes very low.

The basic idea behind the Multigrid method is to take advantage of the fact that low-frequency error components of a fine grid become high-frequency error components in a coarser grid [17]. Thus, the strategy is to move down to a coarser grid once the convergence rate at the current grid saturates. We can think of this as a low-pass filter whose cutoff frequency can vary. That is, once the high-frequency components are filtered out, the cutoff frequency can be further moved down so that the error components which could not be filtered out in the previous setting can be filtered. Thus, a high convergence rate can be sustained by moving through different grid levels.

The basic Multigrid scheme forms a V-cycle in which the downward path computes the residual error while the upward path is the correction path which updates the old estimation with a new approximation. The algorithm of the V-cycle in a recursive form is as follows [17] :

- $\mathbf{A}^h$ is a coefficient matrix at grid level h.
- $\mathbf{f}^h$ is a vector of the values of $f(x, y)$ at grid level h.
- $\mathbf{v}^h$ is a vector of the unknown variable approximations at grid level h.
- I_h^{2h} is an interpolation function mapping from a fine grid to a coarse grid. (Also called *restriction.*)
- I_{2h}^h is an interpolation function mapping from a coarse grid to a fine grid.

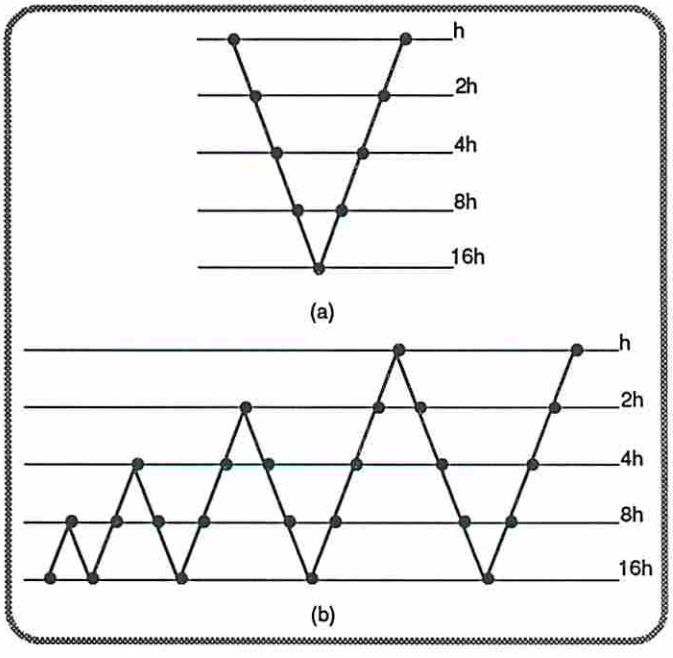

Figure 3.1: The top figure shows the V-cycle and the bottom figure shows the FMV-cycle.

Algorithm MV : $\mathbf{v}^h \leftarrow MV^h(\mathbf{v}^h, \mathbf{f}^h)$

1. Relax ν_1 times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ with initial guess $\mathbf{v}^h$.
2. If Ω^h is Ω^H (coarsest grid) then go to 4.
   Else $\mathbf{f}^{2h} \leftarrow I_h^{2h}(\mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h)$
   $\mathbf{v}^{2h} \leftarrow 0$ (zero as initial guess, for error)
   $\mathbf{v}^{2h} \leftarrow MV^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h})$
   End if
3. Correct $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h\mathbf{v}^{2h}$.
4. Relax ν_2 times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ with $\mathbf{v}^h$ as initial guess.

A more efficient multigrid scheme called the *full multigrid* (FMV) computes the initial guess of the finest level by performing a V-cycle at every grid level using the corrected value of v from the coarser level as the new initial guess. Its algorithm in recursive form is as follows [17] :

Algorithm $FMV : \mathbf{v}^h \leftarrow FMV^h(\mathbf{v}^h, \mathbf{f}^h)$

1. If Ω^h is Ω^H then go to 3.
   Else $\mathbf{f}^h \leftarrow I_h^{2h}(\mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h)$
   $\mathbf{v}^{2h} \leftarrow 0$
   $\mathbf{v}^{2h} \leftarrow FMV^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h})$
   End if
2. Correct $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h\mathbf{v}^{2h}$.
3. $\mathbf{v}^h \leftarrow MV^h(\mathbf{v}^h, \mathbf{f}^h)$ ν_0 times.

3.1.1 The Model Problem

The model problem used in the experiment is the following two-dimensional self-adjoint elliptic equation in the unit square with proper Dirichlet (static) boundary conditions [14]:

$$\begin{aligned} -\mathbf{div}\ (\mathrm{k}\ \mathbf{grad}\ \mathrm{u}) &= \mathrm{f}\ \text{ in } \Omega = [0, 1.0] \times [0, 1.0] \\ \mathrm{u} &= \mathrm{g}\ \text{ on } \partial\Omega \end{aligned}$$

The second equation refers to boundary condition, *i.e.*, on the boundary of the region in question, $u(x, y) = g(x, y)$ where $g(x, y)$ is a known function. Two different values for the diffusion function $k(x, y)$ are used. The first case is $k(x, y) = 1$ and the second case is $k(x, y) = e^{(x+y)}$. When the diffusion function is 1 as in the first case, the well-known Poisson's equation results :

$$-[\frac{\partial^2 u(x, y)}{\partial x^2} + \frac{\partial^2 u(x, y)}{\partial y^2}] = f(x, y)$$

In general, the resulting equation is of the following form :

$$-\frac{\partial}{\partial x}[k(x, y)\frac{\partial u(x, y)}{\partial x}] - \frac{\partial}{\partial y}[k(x, y)\frac{\partial u(x, y)}{\partial y}] = f(x, y)$$

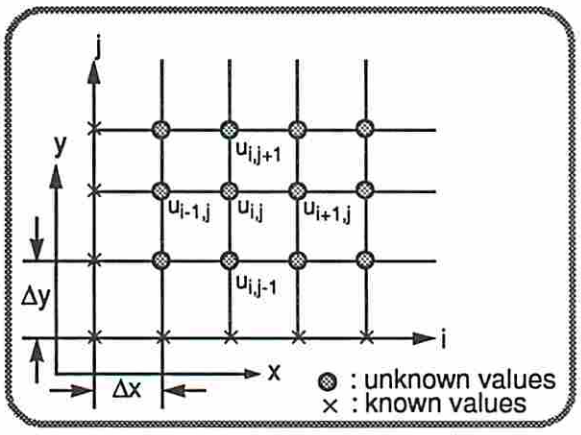

Figure 3.2: The finite difference method discretizes a continuous region into many grid points by dividing the region of interest into equal grid sizes. Note that the known values are at the boundary of the region Ω.

In order to solve the problem numerically, the continuous partial differential equation needs to be discretized. In other words, solutions of the dependent variables are determined only at discrete points within the problem domain even though the variables vary continuously throughout the domain. In the experiment, the partial differential equation is discretized using the finite difference method. The finite difference method is based on a Taylor series expansion in which the order of the *truncation error* depends on the number of terms selected from the Taylor series [89]. In the experiment, a second order approximation is used, that is, the truncation error is of order $O((\Delta x)^2, (\Delta y)^2)$ where Δx and Δy are the grid spaces in the x and y axis, respectively (Figure 3.2). If $\Delta x = \Delta y = h$, an unknown variable u at discrete point x_i and y_j (when $k(x, y) = 1$) can be approximated as in the following equation :

$$u_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + h^2 f_{i,j}}{4} \qquad 0 \leq i \leq N, \quad and \quad 0 \leq j \leq M$$

The variables N and M are the number of grid points in the x and y directions, respectively. The $u_{i,j}$ and $f_{i,j}$ represent the variables $u(x, y)$ and $f(x, y)$ at discrete grid points x_i and y_j.

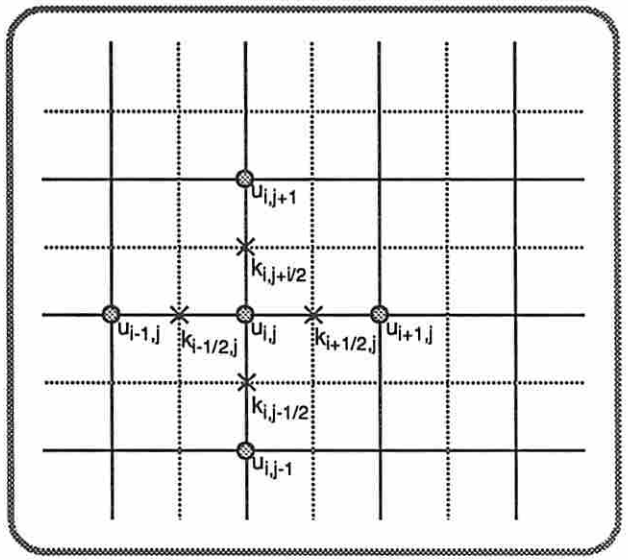

Figure 3.3: When the diffusion function is not constant, using a staggered grid scheme results in a symmetric coefficient matrix.

When $k(x, y)$ is a function of x and y, the *staggered grid* method is used so that the resulting coefficient matrix is symmetric (Figure 3.3) and the $O(h^2)$ is retained. The resulting difference equation looks like the following:

$$u_{i,j} = \frac{k_{i-1/2,j}u_{i-1,j} + k_{i+1/2,j}u_{i+1,j} + k_{i,j-1/2}u_{i,j-1} + k_{i,j+1/2}u_{i,j+1} + h^2 f_{i,j}}{k_{i-1/2,j} + k_{i+1/2,j} + k_{i,j-1/2} + k_{i,j+1/2}}$$

Once the partial differential equation has been discretized, a system of linear equations results which can be written in a vector form as shown below :

$$\mathbf{A}_{(n\times n)}\mathbf{u}_{(n\times 1)} = \mathbf{f}_{(n\times 1)}$$

- **A** is the coefficient matrix. Its size is $n \times n$ and it has the characteristics of being sparse and symmetric.
- **u** is a vector of unknown variables.
- **f** is a vector of the values of $f(x, y)$ at discrete points.

3.1.2 Implementation

In the experiment, a full multigrid algorithm was implemented. As shown in Figure 3.1 (b), this method starts out at the coarsest grid in which each grid point is computed to an exact value. These grid points are then interpolated to the next finer grid. Then a V-cycle is performed on these interpolated grid points. This repeats at every grid level until the finest grid level is reached. This section describes the SISAL implementation of the algorithm and various functions performed as part of the multigrid operations.

The function `MultiV` shown in Figure 3.4 is a SISAL implementation that performs a V-cycle. The function is written in a recursive style and closely resembles the Algorithm *MV* description. It has five input parameters and one output parameter which is a two-dimensional array. The data type `TwoDim` is a user-defined data type which is really a two-dimensional array of double-precision floating point numbers. The first three input parameters `N,n1,n2` of type `integer` are the grid size, the number of relaxations in the downward V-cycle, and the number of relaxations in the upward V-cycle, respectively. Among the two input parameters of type `TwoDim`, `V` is the current approximation of $\mathbf{u}$ and `F` is $\mathbf{f}$ in the equation $\mathbf{Au} = \mathbf{f}$.

The function `MultiV` is called by the function `FMV` which performs the FMV-cycle. In an FMV-cycle, the computation starts from the coarsest grid level and moves up one level at a time. At each higher grid level, a V-cycle is performed (Figure 3.1 (b)). Figure 3.5 is a SISAL implementation of the function `FMV`. This function is also written in a recursive style and closely resembles the *FMV* algorithm described in the previous section.

The functions `MultiV` and `FMV` call the following five functions which are the core functions of the multigrid algorithm:

`Relax` : One of the iterative methods such as Jacobi, Gauss-Seidel, etc. This function is discussed in more detail in subsequent paragraphs.

`ComputeRes` : This function computes *residual* $\mathbf{r}$, that is, $\mathbf{r} = \mathbf{f} - \mathbf{Av}$. It contains (two-level) nested forall loops only.

```
function MultiV (N,n1,n2:integer; V,F : TwoDim returns TwoDim)
  let
    NewGrid := Relax(N,n1,V,F);
    UpdateGrid := if N = 2 then
      Relax(N,1,NewGrid,F)
    else
      let
        Residue := ComputeRes(N,F,NewGrid);
        CoarseF := Restrict(N/2,Residue);
        CoarseErrorG := MultiV(N/2,n1,n2,InitVal(N/2),CoarseF);
        ErrorGrid := InterP(N,CoarseErrorG);
        CorrectedV := Correction(N,NewGrid,ErrorGrid)
      in
        Relax(N,n2,CorrectedV,F)
      end let
    end if
  in
    UpdatedGrid
  end let
end function
```

Figure 3.4: The function `MultiV` is a SISAL implementation of a recursive algorithm which performs a V-cycle.

```
function FMV (N,n1,n2:integer; V,F:TwoDim returns TwoDim)
  let
    Grid := if N = 2 then
      ExactSolve(F)
    else
      let
        Residue := ComputeRes(N,F,V);
        CoarseF := Restrict(N/2,Residue);
        CoarseUpdatedV := FMV(N/2,n1,n2,BndVal(N/2),CoarseF);
        UpdateV := InterP(N,CoarseUpdatedV);
        CorrectedV := Correction(N,V,UpdatedV)
      in
        MultiV(N,n1,n2,CorrectedV,F)
      end let
    end if
  in
    Grid
  end let
end function
```

Figure 3.5: The function `FMV` is a SISAL implementation of a recursive algorithm which performs a FMV-cycle.

`Restrict` : This function performs an interpolation from a grid of size N to a grid of size $N/2$. It is used in the downward path of the V-cycle and contains nested forall loops.

`InterP` : This function performs an interpolation from a coarse grid of size $N/2$ to a fine grid of size N. This function also contains only forall loops.

`Correction` : This function modifies the previously approximated unknown variables by adding the correction values. This function contains forall loops.

Two points were considered in deciding the kind of iterative method to be used for relaxation. The first consideration is the convergence rate. As discussed previously, SOR performs the best if ω_{opt} can be determined. Since this value cannot be determined in general, the next best choice is the Gauss-Seidel iteration. The second consideration is the amount of parallelism available in the algorithm. In this respect, the Jacobi method has the most parallelism. In a Jacobi iteration, the new approximation of a grid point is only a function of the grid points from previous approximations. Therefore, all grid points can be updated in parallel. In the Gauss-Seidel method, on the other hand, a new approximation of a grid point depends partly on the most recently approximated grid points. Due to this data dependency in the algorithm, the Gauss-Seidel method is inherently sequential.

Fortunately, a parallel version of the Gauss-Seidel method exists. It is called the *Red-Black* Gauss-Seidel method [79] and is shown in Figure 3.6. This method is not fully parallel as the Jacobi method. Instead, grid points are updated in two sequential steps. That is, one half of the grid points are updated first and the other half are updated next. At each step, however, grid points can be updated in parallel. Although the amount of parallelism available in the Red-Black Gauss-Seidel method is only half that of the Jacobi method, its superior convergence rate (twice faster than Jacobi) makes it a better iterative scheme (Figure 3.7).

3.1.3 Experimental Results

In this section, we first discuss the performance issue by comparing the execution time of three multigrid programs written in C, FORTRAN, and SISAL. We first

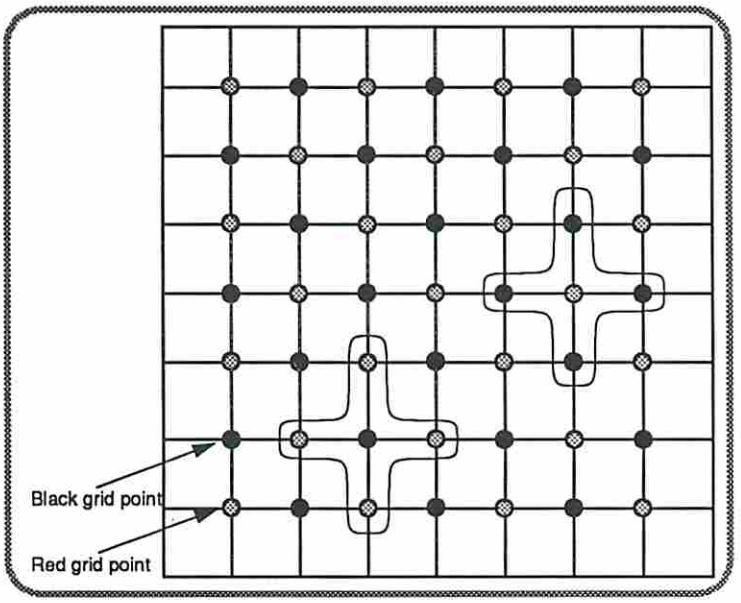

Figure 3.6: The Red-Black Gauss-Seidel is a parallel version of the otherwise sequential Gauss-Seidel iterative method. Notice the way the red and the black grid points are divided. The cross-like regions represent the 5-point stencil used in the approximation.

compare the execution time on a sequential machine and then on a multiprocessor. We then discuss the programmability issue by presenting the approximate development time of each PDE solver.

3.1.3.1 Parallelism Profile of the Multigrid Implementation

For performance measurements, an implementation of a full multigrid algorithm is used. In the program, the diffusion function of $k(x, y) = e^{(x+y)}$ is used. Throughout the measurements, the number of relaxations ν_1 in the downward path of the V-cycle is set to two and the number of relaxations ν_2 in the upward path of the V-cycle is set to one.

The relaxation operation along with other operations such as interpolation, restriction, residue calculation and error correction is performed at every grid level. Althouth there can be many variations of moving around different grid levels, they are all based on the V-cycle. Figure 3.8 shows the parallelism profile of the V-cycle for an 8 by 8 grid using the Red-Black Gauss-Seidel iteration as the relaxation scheme. The 2 by 2 grid is the coarsest grid in the V-cycle. The

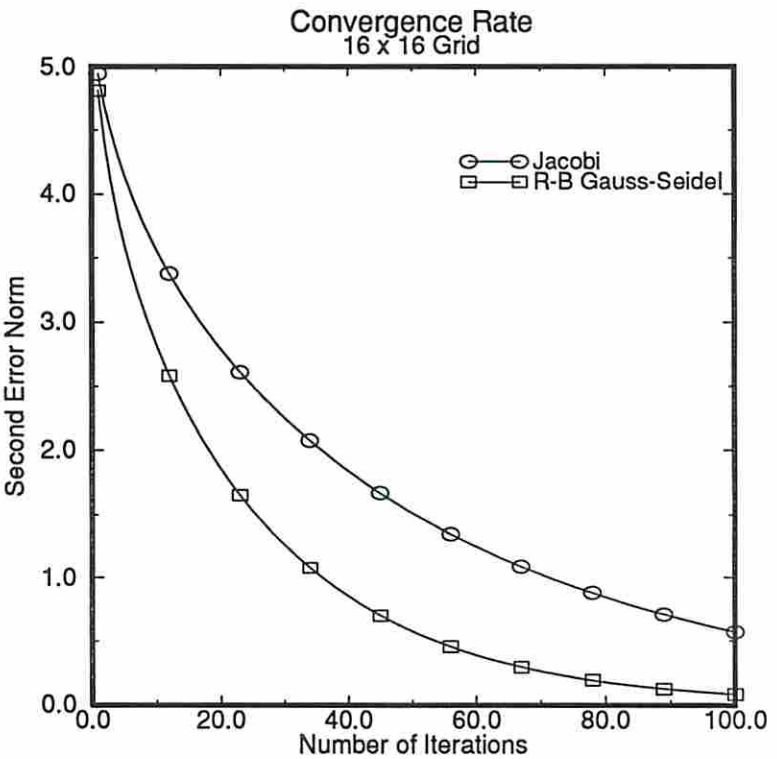

Figure 3.7: Red-Black Gauss-Seidel iteration has a convergence rate superior to that of the Jacobi iteration.

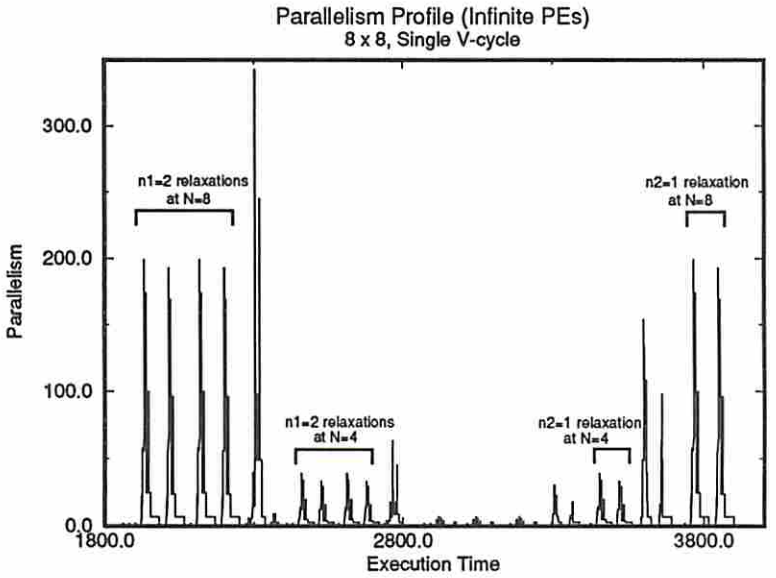

Figure 3.8: The ideal parallelism profile of a single V-cycle where the grid size is 8 by 8. $\nu_1 = 2$ and $\nu_2 = 1$. (See Algorithm MV)

parallelism profile shown is for an ideal case which assumes an infinite number of processors and no communication overhead.

The amount of parallelism is computed by counting the number of executable nodes at each time interval. The nodes are part of the intermediate-level representation of the program which is a directed acyclic graph called Intermediate Form 1 (IF1) [88]. The nodes represent instructions and the edges connecting the nodes represent data dependency relationships among the nodes.

The parallelism profile looks like a cluster of impulses because it is assumed that infinite number of processors is available. That is, all instructions that become executable are assumed to be executed together at the same time. The parallelism profile shows that parallelism decreases by one fourth until the grid size of 2 by 2 is reached and increases back to the original level. The reason the profile is not exactly symmetric is because the relaxation is performed twice going down the grid level ($\nu_1 = 2$), but only once coming up the grid level ($\nu_2 = 1$). Note

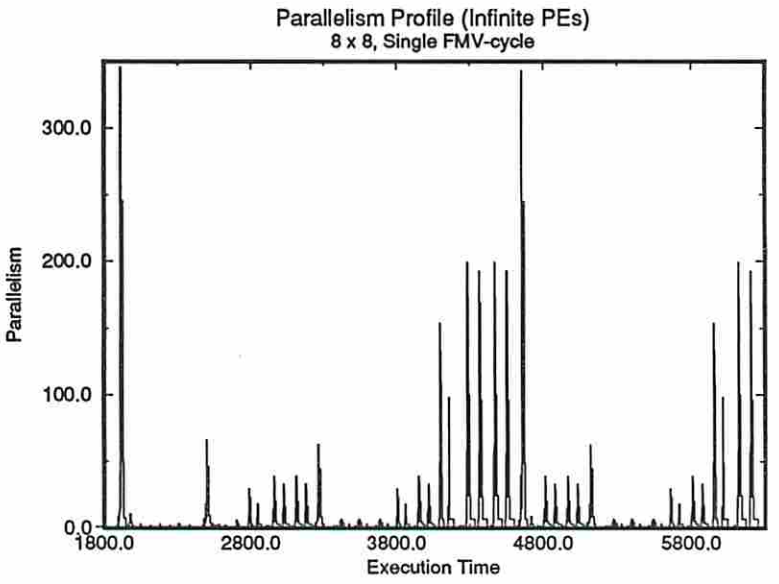

Figure 3.9: The ideal parallelism profile of a single FMV-cycle where the grid size is 8 by 8. $\nu_1 = 2$ and $\nu_2 = 1$. (See Algorithm FMV)

that a single relaxation produces two spikes because in Red-Black Gauss-Seidel iteration, grid points are updated in two sequential steps.

In the current multigrid implementation, a full multigrid V-cycle (FMV-cycle) is used once in the initialization stage followed by regular V-cycles in which the number of repetitions is specified by an input parameter. An FMV-cycle starts from the coarsest grid and moves up to the finest grid. At each grid level, a V-cycle is performed. The parallelism profiles for an FMV-cycle is shown in Figure 3.9 for infinite processors. As expected, we see repeated V-cycle profiles of different sizes. The rightmost pattern is the V-cycle parallelism profile for an 8 by 8 grid.

Figure 3.10 shows the parallelism profile of the full multigrid scheme doing 5 iterations. As mentioned already, the first iteration is the FMV-cycle, and the next 5 iterations are the V-cycles.

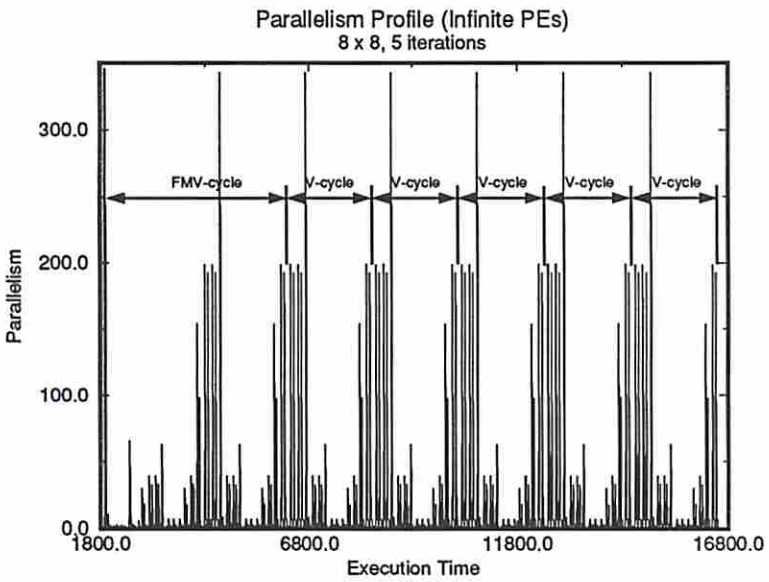

Figure 3.10: The ideal parallelism profile of a full multigrid scheme doing 5 iterations. The grid size is 8 by 8. $\nu_1 = 2$ and $\nu_2 = 1$.

Sun SPARCsystem 400 (64 MBytes) Execution Time (sec.)			
Prob. size	C version	FORTRAN version	SISAL version
128 x 128	5.22	7.61	8.64
256 x 256	21.20	30.94	33.08
512 x 512	88.13	128.38	135.09

Table 3.1: Execution time of the multigrid program written in different languages on a sequential machine.

Sun SPARCsystem 400: Program size (bytes)			
type	C version	FORTRAN version	SISAL version
Object	18,336	13,802	32,515
Executable	49,152	106,496	98,304

Table 3.2: Program size of each version before (object) and after linking (executable).

3.1.3.2 Comparison of the Execution Time

We now compare the execution time of three different implementations of the multigrid PDE solver. In addition to the SISAL version, two different languages were used. One used C in his implementation which we include as one of the samples. The rest were written in FORTRAN and the performance measures we present here are from the fastest FORTRAN version.

We first measured the execution time of the multigrid programs on a sequential machine using a Sun SPARC as the target machine. Table 3.1 shows the execution time of each implementation according to different problem sizes. In the measurement, we had each program execute 5 iterations. Note that by 5 iterations we really mean one FMV-cycle followed by five V-cycles. Among the three programs, the C version resulted in the fastest execution time. This was a surprising result because we expected the FORTRAN version to be the fastest since the FORTRAN language has a better implementation of multi-dimensional arrays than C. It turns out that the C version only utilizes one-dimensional arrays, and through explicit memory management, uses them as two-dimensional arrays. We believe this to be one of the main reasons for its performance. Table 3.2 lists the size of each program compiled for execution on the Sun SPARC machine.

CRAY Y-MP C90 Execution Time (sec.), Prob. size = 512 x 512			
PE	C version	FORTRAN version	SISAL version
1	5.27	30.13	7.49
2	5.27	23.42	4.19
3	5.27	21.14	3.10
4	5.27	19.64	2.51

Table 3.3: Execution time of the multigrid program written in different languages on a multiprocessor CRAY Y-MP C90.

Using the same set of programs, we then measured the execution time on a multiprocessor using a CRAY Y-MP C90 as the target machine. Note that no modification was made to the original programs before compilation. However, all available optimizations were used to the fullest to produce the fastest possible code. The SISAL version of the multigrid program was compiled using the OSC compiler version 12.9.1. The program was compiled with the maximum optimization turned on. In addition to vectorization and concurrentization of loops, the compiler performs a number of memory optimizations to avoid array copying.

The C version of the multigrid program was compiled using the `scc` compiler version 3.0 which is the CRAY Standard C compiler [23]. The maximum optimization level was used which performs aggressive inlining, aggressive autotasking, and aggressive vectorization. In addition, maximum scalar optimization is also performed. Autotasking refers to partitioning of a program such that some portions may be executed concurrently on multiple processors.

The FORTRAN version of the multigrid program was compiled using the `cf77` FORTRAN compiling system [24]. The main components of the `cf77` compiling system are `fpp` (dependency analyzer), `fmp` (translator), and `cft77` (FORTRAN compiler). For our experiment, the `fpp` dependency analyzer plays the most crucial role in the compiling system since no human involvement is allowed in the vectorization/concurrentization process. It alone can determine whether a loop can be vectorized or concurrentized. For example, if `fpp` determines a loop can be safely vectorized, a compiler directive `CDIR@ IVDEP` is inserted which tells the compiler to ignore the apparent data dependency and vectorize the loop.

CRAY Y-MP C90: Program size (bytes)			
type	C version	FORTRAN version	SISAL version
Object	49,240	113,000	79,848
Executable	238,368	748,280	454,008

Table 3.4: Program size of each version before (object) and after linking (executable) compiled on CRAY Y-MP C90.

Table 3.3 shows the execution time of the three multigrid implementations on a CRAY Y-MP C90 with respect to the number of processors. To measure the execution time of the C and FORTRAN versions running on a single processor, the `SECOND ()` function was used. To measure the execution time on multiple processors, `ja` job accounting information facility was used. For the SISAL version, timing facility provided by the run-time system was used. Table 3.4 lists the code size of each version as compiled on the CRAY machine. The problem size of 512 by 512 was used in the measurement. As can be seen from the table, the C version resulted in the fastest execution time when a single processor was used. Once multiple processors are used, however, the SISAL version consistently performed the best. The FORTRAN version performed poorly regardless of the number of processors used. We now report the optimizations performed by each compiler.

CRAY **Standard C compiler** (`scc`):

- All 10 functions are inlined.
- Seven inner `for` loops are vectorized. Among the vectorized loops, three are from the initialization part of the main program. One loop is from function `fset ()` which sets up the grid with initial data. A loop from function `err ()` is vectorized. This function is called every time the highest (finest) grid level is reached. Two loops in function `intadd ()` are vectorized. This function is called at every grid level.
- No loops were concurrentized as can be inferred from Table 3.3.

FORTRAN **compiling system** (`cf77`):

- All six subroutines are inlined.

- Six inner do loops all from subroutines are vectorized. Two loops from subroutine FKU () are vectorized. This routine is called only once in the initialization phase. One loop is vectorized from subroutine INJECT () which is called at every grid level. One loop is vectorized from subroutine INTADD (). This routine is also called at every grid level. One loop is vectorized from subroutine RBGS (). This routine performs the relaxation and it is called at every grid level. One loop is vectorized from subroutine RESERR (). This routine is called only at the highest grid level.

- Three outer do loops are concurrentized through autotasking. They are from subroutines FKU (), INJECT (), and RESERR ().

Optimizing SISAL Compiler (osc):

- Out of 15 functions, 13 functions are inlined. The two that are not inlined are recursive functions MultiV () and FMV ().

- 12 inner loops from 10 functions are vectorized.

- 9 outer loops from 8 functions are concurrentized.

Taking into account the fact that all three programs resulted in similar execution time on a sequential machine, we conjecture that the C and FORTRAN versions were written in a style which makes it difficult for the automatic vectorization/concurrentization compiling system. On the other hand, due to the functional semantics of SISAL, the OSC compiler extracted all existing parallelism in the program and was able to appropriately vectorize and concurrentize.

3.1.3.3 Measurement on other Parallel Machines

We executed the SISAL version of the multigrid PDE solver on two other parallel machines in addition to the CRAY Y-MP C90. They are a Silicon Graphics machine with four processors and a Sequent Balance with 16 processors. The Silicon Graphics machine is based on MIPS R3000 microprocessors while the Sequent Balance uses National Semiconductor's NS32032 microprocessors. Both machines are slow by today's standard. The reason for reporting the performance of the

Execution Time (sec.)		
	512 x 512	256 x 256
PE	SGI	Sequent Balance
1	457.78	3989.20
2	287.14	1993.17
3	192.69	-
4	146.00	1014.61
8	-	516.11
12	-	362.29
16	-	274.89

Table 3.5: Execution time of the SISAL version of a multigrid program on Silicon Graphics and Sequent Balance.

SISAL version on these machines is to show the portability of SISAL and how well the compiler parallelizes the code[1]. Table 3.5 shows the execution time. Note that smaller problem size was used on Sequent Balance due to slow speed of the machine.

The speedup curves of the SISAL version attained from the CRAY, Silicon Graphics, and Sequent Balance machines are shown in Figures 3.11 and 3.12. The graphs indicate that the OSC compiler was able to extract the existing parallelism from the program and appropriately vectorize/concurrentize for the target machines. Although the execution time on the Silicon Graphics and Sequent Balance is slow, the speedup curve is consistent with that from the CRAY machine.

3.1.3.4 Effects of Memory Optimization

The execution time of the SISAL multigrid program discussed in the previous section was compiled with all the memory optimizations enabled. To observe the effects of memory optimizations, we have compiled the same multigrid program into three different versions. They are,

1. Fully optimized : OPT

2. Only build-in-place optimization : OBIP

[1]These machines do not have hardware vector facilities.

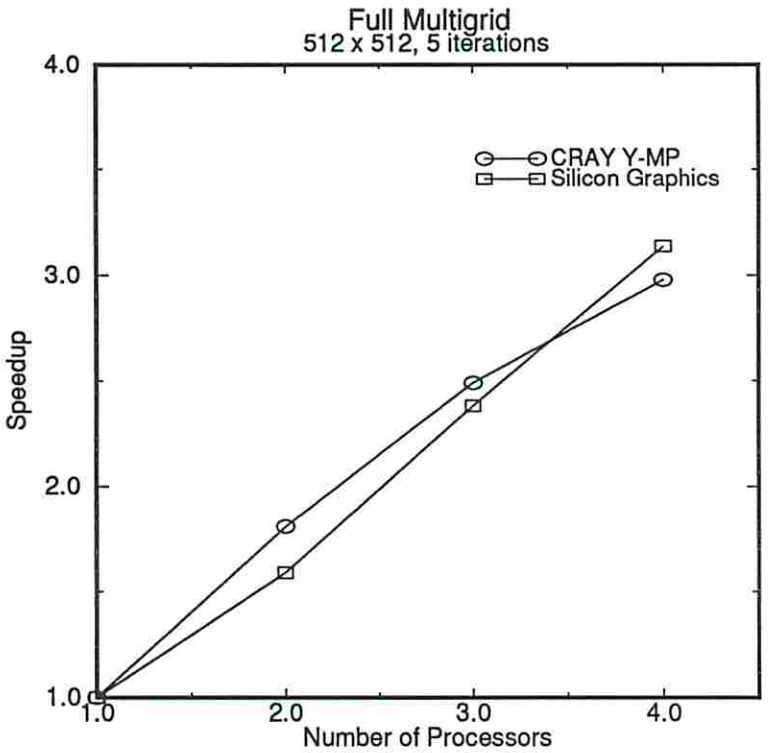

Figure 3.11: Speedup attained by executing the SISAL version of a full multigrid PDE solver on the CRAY Y-MP C90 and Silicon Graphics machines. The problem size is 512 by 512.

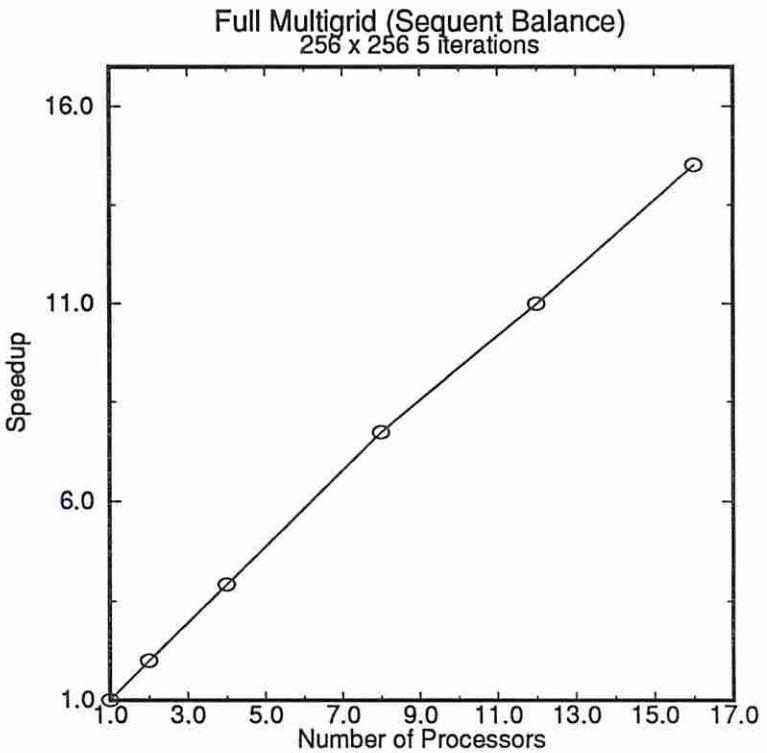

Figure 3.12: Speedup attained by executing the SISAL version of a full multigrid PDE solver on a Sequent Balance. The problem size is 256 by 256.

3. Only update-in-place optimization : OUIP

Table 3.6 shows the execution times resulting from different versions of the same multigrid program. The execution times shown are from a CRAY-2S running a problem size of 128 by 128. While the execution time of the OPT version scales with respect to the number of available processors, the OUIP version hardly scales up. Furthermore, compared to the optimized version, the execution time is longer by a factor ranging from 5.7 (1 PE) to 13.82 (4 PES). The OBIP version scales slightly, but its execution time is even longer than that of the OUIP version. Compared to the OPT version, its execution time is longer by a factor ranging from 15.16 (1 PE) to 25.30 (4 PES). Note that the execution time shown in the table is for a problem size that is one eighth of the size used in the previous section for performance comparison.

We now look at the number of failed build-in-place (BIP) and update-in-place (UIP) operations as well as the total number of bytes copied as a result (Table 3.7). The OPT version has no failed BIP or UIP operations resulting in zero copying. The OUIP version fails all of the 44,116 attempted build-in-place operations resulting in approximately 21 MBytes of copying. The OBIP version fails all of the 1,018 attempted update-in-place operations resulting in approximately 700 KBytes of copying.

Note that among the two versions that resulted in copying, the OUIP version has more copying, yet actually has a better execution time! To explain this phenomenon we need to analyze the effects of memory optimization on concurrentization and vectorization. In the multigrid program compiled with only update-in-place optimization, virtually all loops are sequentialized. Only one nested loop in function `ComputeDiff` is concurrentized and vectorized. In the OBIP version, all outer loops are concurrentized and four inner loops are vectorized. Table 3.8 shows the concurrentization and the vectorization status of each version. Note that the function names `InterP` and `rbGS` are repeated in the first column of the table. This is because there are two separate parallel loops executing in sequence.

Among the four vectorized loops in the OBIP version, two (`SetInitVal` and `SetBoundVal`) are non-compute intensive loops which simply initialize an array to zero. The other two loops are only executed once in the initialization stage. Thus,

Execution Time (sec.), 128 x 128			
PE	OPT	OUIP	OBIP
1	1.62	9.24	24.56
2	0.96	9.16	19.50
3	0.78	9.36	18.93
4	0.71	9.81	17.96

Table 3.6: The execution time of a same program can vary greatly depending on the level of memory optimizations applied during compile-time.

	Failed BIP Ops.	Failed UIP Ops.	Bytes Copied
OPT	0	0	0
OUIP	44,116	0	21,942,808
OBIP	0	1,018	707,184

Table 3.7: The table shows the number of failed build-in-place (BIP) and update-in-place (UIP) operations when different levels of optimization are performed.

although four loops are vectorized in the OBIP version, no vectorization effect is visible. On the other hand, the only loop that is concurrentized and vectorized in the OUIP version is executed at every iteration and at every grid level. Therefore, despite the fact that only one loop is vectorized, the vectorization effect is visible.

3.1.3.5 Programmability

Table 3.9 shows the development time of every PDE solver implemented in the experiment. The development time shown in the second column of the table includes the time spent in understanding the algorithms as well as the days spent on actual program development. On the average, half of the time was devoted to understanding of the algorithm and the mathematical backgrounds while the other half was spent on the actual implementation. In the case of the multigrid PDE solver, approximately two weeks were spent in understanding the algorithm and one week on actual program development.

In the experiment, the preassigned development time was just enough for everyone to finish the programs. As discussed in the previous section, programs written in different languages ran with similar speed on a sequential machine.

Concurrentized (Y,N), Vectorized (Y, N)			
Recommended loops (function name)	OPT	OUIP	OBIP
`InterP`	Y,Y	N,N	Y,N
`InterP`	Y,Y	N,N	Y,N
`Restrict`	Y,Y	N,N	Y,N
`ComputeF`	Y,Y	N,N	Y,Y
`rbGS`	Y,Y	N,N	Y,N
`rbGS`	Y,Y	N,N	Y,N
`SetInitVal`	Y,Y	N,N	Y,Y
`SetBoundVal`	Y,Y	N,N	Y,Y
`ComputeRes`	Y,Y	N,N	Y,N
`Correction`	Y,Y	N,N	Y,N
`ComputeExactU`	Y,Y	N,N	Y,Y
`ComputeDiff`	Y,Y	Y,Y	Y,N

Table 3.8: Status of the loops recommended for vectorization on a CRAY Y-MP.

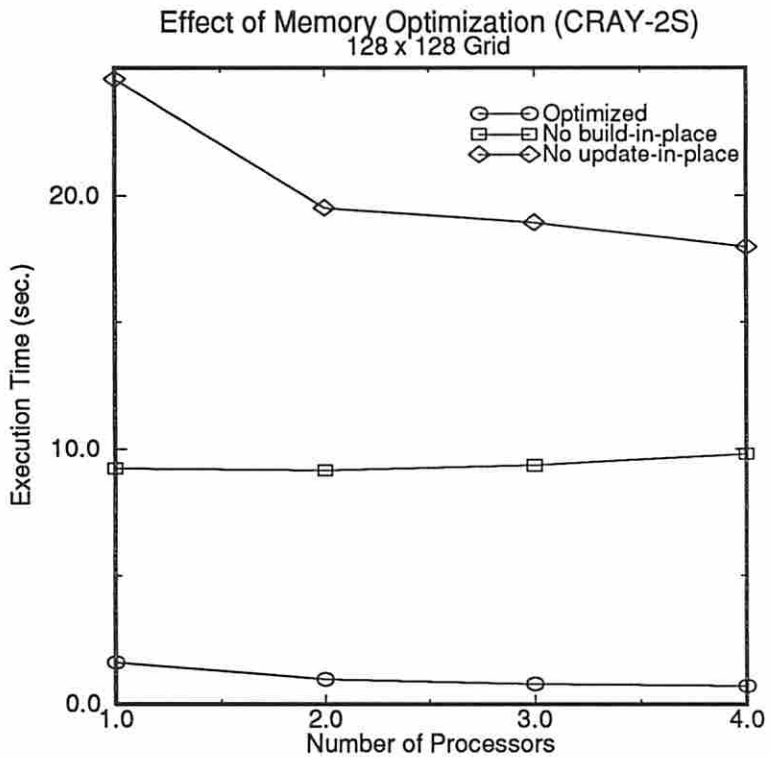

Figure 3.13: This graph shows that memory optimization has a significant effect on the execution time.

Solvers	Develop. Time (days)
Basic iterative methods	29
Multigrid	23
Precond. Conj. Gradient	21
Domain Decomposition	30

Table 3.9: The time spent to learn and implement each PDE solver presented in the course.

However, the programs written in C and FORTRAN did not perform well on parallel machines. The CRAY C compiler did a good job of vectorizing the code, but it could not concurrentize any of the loops in the C program. The FORTRAN compiler neither vectorized nor concurrentized well. This probably is not the fault of the compilers, but the way the programs were written.

To improve the performance of the programs written in C and FORTRAN on parallel machines, more time is required to fine-tune the program. This may entail modifying the program itself and/or inserting necessary directives indicating the parts to be vectorized or concurrentized. On the other hand, the same SISAL program written without any assumption on the target architecture resulted in respectable performance both on a sequential machine as well as parallel machines.

If the programmability issue is addressed in terms of running the programs on sequential machines, this experiment showed that the imperative programming style is sufficient. In terms of parallel computing, on the other hand, the experiment result seems to favor the functional programming style.

3.2 Case Study 2: Domain Decomposition

Domain decomposition is a general terminology used for methods that partition a region of interest into multiple subregions such that each subregion can be computed in parallel or semi-parallel fashion. Semi-parallel means that some interworking between subregions may be necessary. This is useful when the amount of computations required on the main grid differs greatly from one region to another. Domain decomposition is also useful on sequential machines. By operating on one

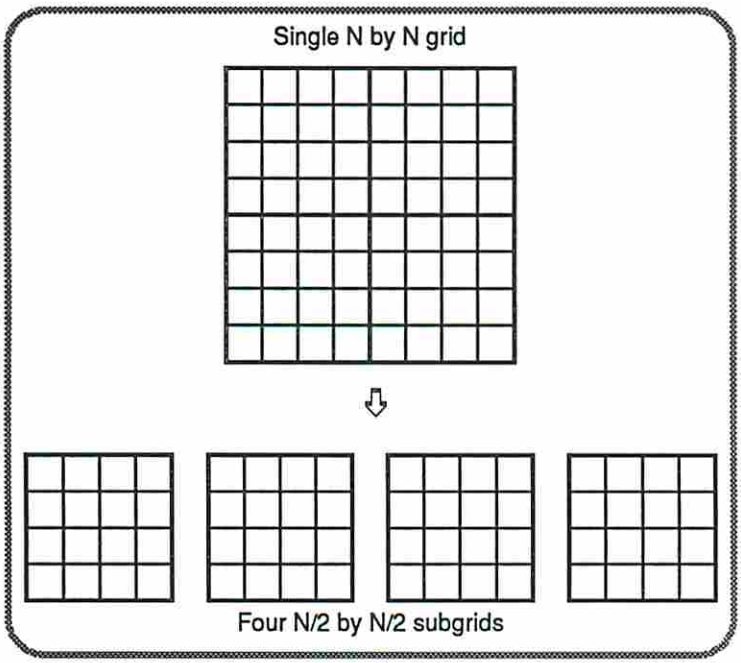

Figure 3.14: Through domain decomposition, a computation region can be subdivided into multiple regions so that each subregion can be computed in parallel.

subregion at a time, memory requirement can be reduced significantly. Figure 3.14 shows a grid is divided into four subgrids.

In this section, we discuss an implementation of a particular domain decomposition called the Symmetric Domain Decomposition (SDD) [80]. The main advantage of this method is that once a grid is divided into subregions, each subregion can be computed independently from other subregions. Once all the subregions are computed, they are merged to generate the solutions for the original grid. In our implementation, we use the same problem discussed in the multigrid implementation. Furthermore, the same multigrid algorithm is used to compute the subregions.

3.2.1 Symmetric Domain Decomposition Method

The Symmetric Domain Decomposition method we describe here is based on the same model problem used in the multigrid section. To make notations simpler, we assume the diffusion function $k(x, y) = 1$ resulting in the following equation.

Because the diffusion function is constant, we use Δ to represent the differential operator $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$.

$$\begin{aligned} -\Delta u &= f \text{ in } \Omega = [0, 1.0] \times [0, 1.0] \\ u &= g \text{ on } \partial\Omega \end{aligned}$$

When the model problem with Dirichlet boundary conditions is partitioned into four subgrids according to the SDD algorithm, the original equations shown above is transformed into the following four sets of equations in $\Omega_{11} = [0, 0.5] \times [0, 0.5]$:

Subregion 1:
$$-\Delta u^{++} = f^{++} \text{ in } \Omega_{11}$$
$$u^{++} = g^{++} \text{ on } \Gamma_{11},\ D_x u^{++} = 0 \text{ on } \Sigma_1,\ D_y u^{++} = 0 \text{ on } \Sigma_2$$

Subregion 2:
$$-\Delta u^{+-} = f^{+-} \text{ in } \Omega_{11}$$
$$u^{+-} = g^{+-} \text{ on } \Gamma_{11},\ D_x u^{+-} = 0 \text{ on } \Sigma_1,\ u^{+-} = 0 \text{ on } \Sigma_2$$

Subregion 3:
$$-\Delta u^{-+} = f^{-+} \text{ in } \Omega_{11}$$
$$u^{-+} = g^{-+} \text{ on } \Gamma_{11},\ u^{-+} = 0 \text{ on } \Sigma_1,\ D_y u^{-+} = 0 \text{ on } \Sigma_2$$

Subregion 4:
$$-\Delta u^{--} = f^{--} \text{ in } \Omega_{11}$$
$$u^{--} = g^{--} \text{ on } \Gamma_{11},\ u^{--} = 0 \text{ on } \Sigma_1,\ u^{--} = 0 \text{ on } \Sigma_2$$

The symbol Γ_{11} refers to the boundary regions $\{0 \leq x \leq 0.5, y = 0\} \cup \{x = 0, 0 \leq y \leq 0.5\}$. The symbol Σ_1 refers to the boundary region $\{x = 0.5, 0 \leq y \leq 0.5\}$ and the symbol Σ_2 refers to the boundary region $\{0 \leq x \leq 0.5, y = 0.5\}$. Note that some subgrids now have Neumann boundary conditions. Subregion 1 has Neumann boundary conditions on Σ_1 and Σ_2. Subregions 2 and 3 have Neumann boundary conditions on Σ_1 and Σ_2, respectively. Subregion 4 only has Dirichlet boundary conditions.

The functions $f(x, y)$, and $g(x, y)$ of the original problem are transformed according to the following formulas:

$$h^{++}(x, y) \equiv (h(x, y) + h(1 - x, y) + h(x, 1 - y) + h(1 - x, 1 - y))/4$$

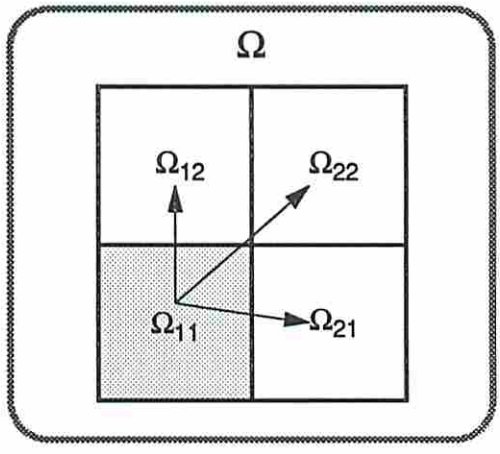

Figure 3.15: After each subregions are computed on Ω_{11}, grid points of the subregions need to be combined and transformed to the rest of the points of the original domain $\Omega = \Omega_{11} \cup \Omega_{12} \cup \Omega_{21} \cup \Omega_{22}$.

$$
\begin{aligned}
h^{+-}(x,y) &\equiv (h(x,y) + h(1-x,y) - h(x,1-y) - h(1-x,1-y))/4 \\
h^{-+}(x,y) &\equiv (h(x,y) - h(1-x,y) + h(x,1-y) - h(1-x,1-y))/4 \\
h^{--}(x,y) &\equiv (h(x,y) - h(1-x,y) - h(x,1-y) + h(1-x,1-y))/4
\end{aligned}
$$

Note that the four subregions computed on $\Omega_{11} = [0, 0.5] \times [0, 0.5]$. Therefore, once each subregion is computed, the grid points belonging to each subgrid need to be combined and transformed to the grid points of the original domain $\Omega = [0, 1.0] \times [0, 1.0]$ (Figure 3.15). The merging and transformation of grid points from subregions to the original domain is defined by the following equations:

$$
u(x,y) = \begin{cases}
u^{++} + u^{+-} + u^{-+} + u^{--}(x,y), & (x,y) \in \Omega_{11} \\
u^{++} - u^{+-} + u^{-+} - u^{--}(x,1-y), & (x,y) \in \Omega_{12} \\
u^{++} + u^{+-} - u^{-+} - u^{--}(1-x,y), & (x,y) \in \Omega_{21} \\
u^{++} - u^{+-} - u^{-+} + u^{--}(1-x,1-y), & (x,y) \in \Omega_{22}
\end{cases}
$$

3.2.2 Experimental Results

In the SISAL implementation of the SDD method, each subdomain is assigned to a processor and within each processor, a subgrid is computed according to the

FMV multigrid algorithm discussed previously in the chapter. Figure 3.16 shows the parallelism profile of the SDD method when the grid size is 8 by 8. Table 3.10 shows the execution time and the memory usage of the Symmetric Domain Decomposition method according to the number of processors used. Based on the experiment, the execution time of the SDD implementation is inferior to that of the multigrid's. The only advantage is in the memory usage. The amount of memory used by the SDD implementation grows as more processors are used. The amount of memory used in the multigrid method is constant with respect to the number of processors used.

There are a number of reasons why the SDD method resulted in inferior performance. One reason is due to the run-time execution model employed by the OSC compiler. Concurrent execution of loops in OSC is based on a *worker* model which has a dynamic load balancing feature. In this model, a worker process is created on each processor when a program is first loaded for execution. Each worker process gets work by accessing the ready task queue located in the shared memory. The task queue is continually accessed by the worker processes until no more work is available.

This execution model has the same effect of domain decomposition by dynamically partitioning the outer loop of the multigrid implementation. On the other hand, the SDD implementation is statically partitioned at the algorithm level where each two-dimensional subgrid is executed in sequential fashion at each processor. The SDD implementation has added overhead from sequential loops and two-dimensional array handling which requires several levels of pointer chasing. The advantage of the dynamic domain decomposition effect can best be seen when three processors are used. Table 3.10 shows that while the execution time of the multigrid implementation increases from two processors to three processors, the execution time of the SDD implementation stays the same.

Another reason comes from using multigrid method as the basic solver. As discussed, multigrid method is considered a fast solver meaning the order of computation complexity is independent of the problem size. Thus reducing the problem size into multiple subgrids does not affect the number of required iterations.

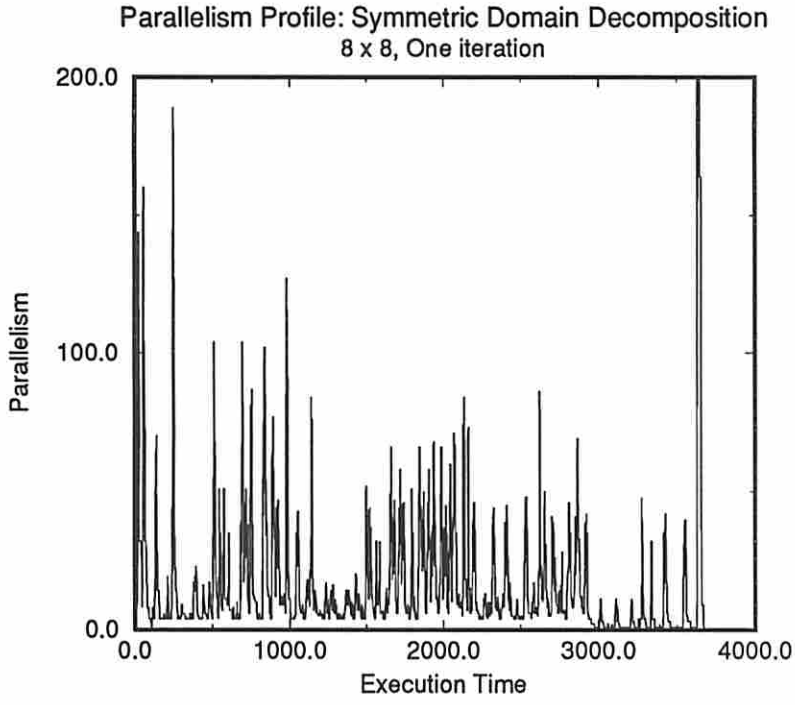

Figure 3.16: The ideal parallelism profile of the Symmetric Domain Decomposition method doing 1 iteration. The grid size is 8 by 8.

CRAY Y-MP C90, Prob. size = 512 x 512				
	SDD		Multigrid	
PE	Exec.Time	Mem.Usage	Exec.Time	Mem.Usage
1	3.95(sec)	17.3(Mbyte)	1.44(sec)	30.4(Mbyte)
2	2.24	20.5	0.92	30.4
3	2.24	28.8	0.74	30.4
4	1.19	35.7	0.65	30.4

Table 3.10: The execution time of the SISAL implementation of the Symmetric Domain Decomposition method and the Full Multigrid method on a multiprocessor CRAY Y-MP C90.

The first working version of the SDD method was completed in approximately two weeks once the algorithm was clearly understood. A number of different versions were written to improve the performance. We found there was no significant difference in performance between different versions of the method. Unlike the multigrid case, no specific comparison can be made with respect to other implementations using imperative languages because there were no counterparts for the implementation of the SDD method.

3.3 Conclusions

Through an experiment, we have shown that functional programming is indeed a viable approach to parallel computing providing programmability, performance, and portability. The SISAL program, which has originally been written for a sequential machine, efficiently executed on a number of parallel machines without requiring the programmer to manually parallelize the code. The implicit parallelism of SISAL, therefore, has allowed the programmer to concentrate on the implementation of the algorithm without having to worry about low-level execution details. Once a program is verified to work correctly on a sequential machine, it can be run on various parallel machines without program modification. While the programs written in C and Fortran performed slightly better than the SISAL version on a sequential machine, they resulted in inferior performance on a multiprocessor.

However, the current version of OSC is tailored for execution on *shared global address space* machines which currently employ a relatively small number of processors (< 30). On the other hand, a growing number of parallel machines have already been introduced or are being introduced which employ a large number of processors. Logically, some machines have a shared global address space and some do not. Physically, however, all large machines have distributed memory spread across the processors which makes memory accesses nonuniform and latency a serious issue to consider. To achieve good performance on such machines, it is imperative to mask the communication overhead by efficiently overlapping

computation and communication. The multithreaded execution model [42] utilizing a finer grain parallelim than the current OSC is one candidate for the next generation of the SISAL compiler.

Chapter 4

Hierarchical Activation Management Technique

This chapter presents a new activation management technique which aims to reduce the frequency of expensive activation context switches. This is achieved by making it possible to activate multiple instances of a code block on a single activation frame. This in effect, multiplies the number of active threads resulting in longer resident time of an activation on a processor. Based on this technique, it is possible to exploit parallelism better by reducing context switching overhead. This chapter is based on a work presented in [66].

4.1 Introduction

The data-driven execution model is a promising model for exploiting fine-grain parallelism since the executability of an instruction is only dependent on the availability of the input operands. However, in order to efficiently support such a radically new execution mechanism, special data-flow architectures such as the Manchester machine [83], Sigma-1 [85], Epsilon [49], Monsoon [53], etc. had to be developed [45].

Driven by the high cost of these complex data-flow architectures and their poor performance when faced with sequential codes, the execution model for exploiting fine-grain parallelism has evolved gradually from the data-driven to the multithreaded execution model [45]. The multithreaded execution model can be seen as a pragmatic reincarnation of the data-driven model to better suit conventional processors with minimum architectural extensions.

Logically, we can view the data-driven and the von Neumann execution models as located at the two opposite ends of a spectrum. On this spectrum, the multithreaded execution model can be placed somewhere in the middle. The Hybrid architecture [60], P-RISC [74], and the EM-4 machine [84] are representatives of such an architecture. These architectures resemble conventional RISC processors extended with primitive instructions to support synchronization of data and triggering/merging of threads.

The Threaded Abstract Machine (TAM) has gone a little further in that, through a novel compiler-directed scheduling strategy, efficient fine-grain multithreading can be achieved even on conventional processors with no architectural extensions [25]. The basic objective of the scheduling strategy in TAM is to minimize the context switching overhead by giving higher priority to those threads which belong to the activation that is currently resident on the processor. This is coupled to an efficient communication mechanism which quickly injects into the currently executing activation the data just received [93].

TAM has shown that efficient fine-grain multithreading can be achieved on conventional processors by employing a scheduler whose policy is based on storage hierarchy. However, there are many instances, purely due to the nature of the program, where even such a scheduling policy is not effective. These are usually the inner loops of numerical applications where the loop body is relatively short while the amount of array read operations is disproportionally large. The array read operations are potential remote operations which can incur a large latency. In such cases, there simply may not be enough available threads to mask the latency belonging to the currently resident activation. The processor may then either idle or switch to a new activation. Both cases will decrease CPU utilization.

The objective of our study is to demonstrate a compiler solution which can effectively handle the type of loops described in the previous paragraph. More specifically, we introduce a new activation management technique which dynamically allows multiple iterations to be concurrently active on a single activation frame. It means that multiple loop iteration instances can be treated as a single activation. By allowing multiple iterations to be concurrently resident on a processor, more threads are generated. These threads can be used to mask the

latency and increase the interval between expensive context switches. The execution model proposed does not assume a completely new hardware multithreaded architecture. This means that no hardware built-in scheduling is assumed (such as in Tera, for example [3]). The thread scheduling is assumed to be the responsibility of the compiler as in TAM. However, the proposed scheme requires some minimal architectural support.

4.2 Motivation

In [7], it was argued that because latency is such a serious problem in large multiprocessor environments (much more so than in single processor cases), an inherently latency-tolerant execution model is desirable. The authors argued that arbitrary latency caused by remote read operations can be masked through a split-phase remote read mechanism where a new thread of computation can continue while the remote memory read request is pending. In addition, to synchronize the asynchronously arriving data, a large synchronization name space is also needed. If enough useful work exists, latency should be completely hidden and the processor fully utilized.

However, if the code block of an activation is too short to provide sufficient number of compute threads to mask the latency, more than one activation can be assigned to the same processor so that the processor can be better utilized by executing threads from multiple activations. This is a form of dynamic loop unrolling. We now demonstrate this using `Loop1` which is one of the Livermore Loop kernels written in the SISAL functional language [71]. The Livermore Loops consist of 24 computational kernels found in large scale scientific applications [37]. In the program, the data type `OneDim` is a one-dimensional array of real numbers.

```
function Loop1 (n:integer; Q,R,T:real; Y,Z:OneDim
                returns OneDim)
  for k in 1,n
  returns array of
    Q + (Y[k] * (R * Z[k+10] + T * Z[k+11]))
  end for
```

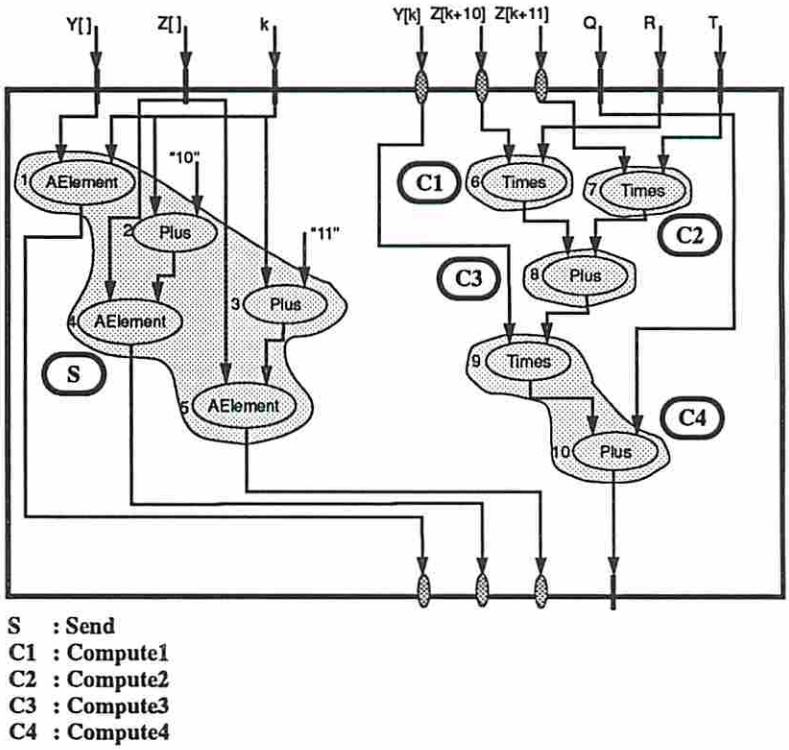

Figure 4.1: Modified IF1 graph of the Livermore Loop 1 with thread partitioning.

```
end function
```

The main body of the loop consists of three array reads and five floating-point operations. Excluding the integer operations used in the array subscript calculation, the array read operations account for approximately 38% of the total. Figure 4.1 shows the modified version of the optimized Intermediate Form 1 (IF1) graph [88] (IF1 graph is a data dependency graph produced by the SISAL frontend). In the graph, the input ports marked with vertical rectangular bars represent input parameters and the ones marked with ovals represent the arriving array elements from split-phase read operations. The ovals in the input ports are matched with the corresponding ovals in the output ports.

From the IF1 graph representation, the loop body of the Livermore Loop 1 is partitioned into five threads. The main criterion of our thread partitioning strategy is to minimize the activation delay of the threads which are triggered by the returning array elements. This strategy was used to maximize the probability

that there will be a ready thread (belonging to the same activation) after the completion of a thread. The nodes constituting each thread are as follows.

1. **Send thread (S)**: transmits read requests for array elements `Y[k]`, `Z[k+10]`, and `Z[k+11]`; IF1 nodes 1,2,3,4,and 5 belong to this thread.

2. **Compute1 thread (C1)**: This thread is triggered by the arrival of `Z[k+10]` and computes the intermediate result `R1[k]` = `Z[k+10]` × `R`; IF1 node 6 belongs to this thread.

3. **Compute2 thread (C2)**: This thread is triggered by the arrival of `Z[k+11]` and computes the intermediate result `R2[k]` = `Z[k+11]` × `T`; IF1 node 7 belongs to this thread.

4. **Compute3 thread (C3)**: This thread is triggered by the termination of the C1 and C2 threads. It adds the two intermediate results produced by the two threads. `R3[k]` = `R1[k]` + `R2[k]`; IF1 node 8 belongs to this thread.

5. **Compute4 thread (C4)**: This thread is triggered by the arrival of `Y[k]` and the termination of the C3 thread. It computes the final value, `Result[k]` = `T` + (`Y[k]` × `R3[k]`); IF1 nodes 9 and 10 belong to this thread.

Using this thread partitioning, two sets of simulations have been performed to observe the effect of latency tolerance with respect to the number of activations. In the simulation, all the elements of the arrays `Y[]` and `Z[]` were assumed to be ready for consumption when the Livermore Loop1 instances are activated. Furthermore, the array elements are distributed across the nodes in an interleaved manner to facilitate parallel array access. The first set of simulation assumes the network latency to be 10 clock cycles while the second set assumes the network latency to be 20 clock cycles. For both simulations, once the array handler receives a read request, an additional memory latency of 10 cycles is incurred assuming the array elements are not cached [1]. An instruction is assumed to be executed

[1] Assuming the clock cycle of the current fastest RISC processor chip is around 5 nsec (200 MHz) and typical DRAM access time is around 60 nsec, this is a reasonable number.

Livermore Loop 1, 64 PEs, Data Size = 3200				
	Network Lat. = 10		Network Lat. = 20	
A/PE	PE Util	Avg.CS/A	PE Util	Avg. CS/A
1	0.577	0.0	0.488	0.0
2	0.720	1.0	0.621	0.9
3	0.781	1.4	0.665	1.3
4	0.810	1.6	0.707	1.5
5	0.820	1.7	0.717	1.7
6	0.833	1.8	0.735	1.9
7	0.831	1.7	0.752	2.0
8	0.837	1.7	0.737	2.1
9	0.855	1.6	0.758	2.2
10	0.861	1.6	0.769	2.2

Table 4.1: The table shows the processor utilization and the average number of context switches per activation with respect to the number of activations per processor: A = Activation, CS = Context Switches

each clock cycle. The number of activations assigned to a processor ranges from one to ten.

Table 4.1 clearly shows that processor utilization increases as more loop instances are activated. As more instances are activated on each processor, more threads are available for computation while waiting for the requested array elements. The table shows that when the network latency is 10 cycles, approximately 96% of the maximum processor utilization is obtained by assigning two activations to each processor. When the network latency is doubled to 20 cycles, the rate of increase in the processor utilization with respect to the number of activations is slowed. This is due to the fact that the computation becomes more influenced by the queueing effect resulting from a slower network. For this case, only 82% of the maximum processor utilization is achieved when two activations are assigned per processor. To reach approximately 94% of the maximum utilization, five activations had to be assigned to each processor.

The improvement in processor utilization obtained by assigning multiple activations to each processor as shown in Table 4.1 is misleading because it ignores the existence of a memory hierarchy. In fact, the same assumption is used in the

Thread Scheduling Policy:

```
while Not Done loop
   if No Arrived Messages then
      if No Compute Thread then
         Context Switch;
      else
         Execute Compute Thread;
      end if;
   else
      Read Data and Execute Corresponding
      Interface Thread;
   end if;
end while;
```

Figure 4.2: This thread scheduling policy switches context to another activation only if no thread is available within the current activation.

argument of [7]. It is well known, of course that most computers employ a hierarchical memory system for cost and performance reasons. For example, registers which are closest to the processor are the fastest memory devices, but they are also the most expensive. Therefore, only a relatively small number of registers can be supported in the processor. To fully utilize the processor, however, a portion of the activation currently being executed must be resident at the highest level of the hierarchy [28].

The consequence of this resource restriction is that only a small fraction of the available work can be resident on the processor at a time and a portion of work must be brought in and out of the processor over the duration of computation, *i.e.*, context switching occurs. Since we assume no hardware support for multithreading, context switching is an overhead which must be minimized. Some multithreaded architectures such as Hep and Tera have hardware scheduling which switches context at each clock cycle [3].

Table 4.1 also shows the average number of context switches per activation. Note that the result is based on the thread scheduling policy shown in Figure 4.2

which maximizes the resident time of an activation once it is brought into the processor. Note that the scheduler switches context only if no message has arrived and no ready threads exist. Nevertheless, due to the small number of threads in the Loop 1 code block, context switching is unavoidable when two or more activations are assigned to a processor. Note that no context switch occurs when there is just a single activation on a processor. For this case, when all the executable threads run out, the processor idle.

From Table 4.1, we also observe that the frequency of context switching is higher for the case of the slower network. This is intuitive since the average arrival rate of data which activate compute threads is lower. The conclusion of this example using the Livermore Loop 1 is that despite the use of a scheduling policy which maximizes the resident time of an activation, a sizeable number of context switches can still occur. Context switching hurts overall performance by reducing processor utilization where the degree of reduction depends on the actual overhead. Figure 4.3 illustrates that in reality, context switching creates overhead which reduces processor utilization. In the figure, simulations were performed with two different values of context switch penalty. It shows that when the context switch penalty is 20 cycles, assigning multiple activations to a processor has negligible improvement due to the context switch overhead.

4.3 Multiple Iterations per Activation Model

This section describes our proposed execution model which allows multiple loop iteration instances to be treated as a single activation followed by the required architectural support.

4.3.1 The Execution Model

We have seen in the previous section that despite a scheduling policy which aims at sustaining the duration of a resident activation, context switches still occur in loops with short loop body. Furthermore, this situation worsens as the system latency increases. This is because the time interval between the moment the remote read requests are sent out and the moment at which the corresponding

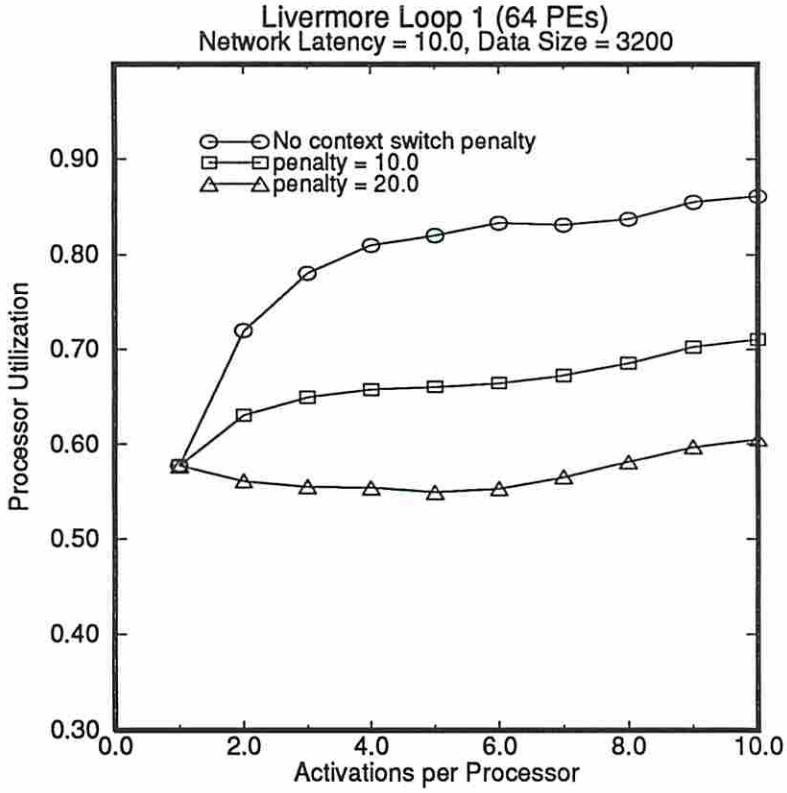

Figure 4.3: The graph shows that the overhead of context switches reduces processor utilization which adversely affects overall performance of a program.

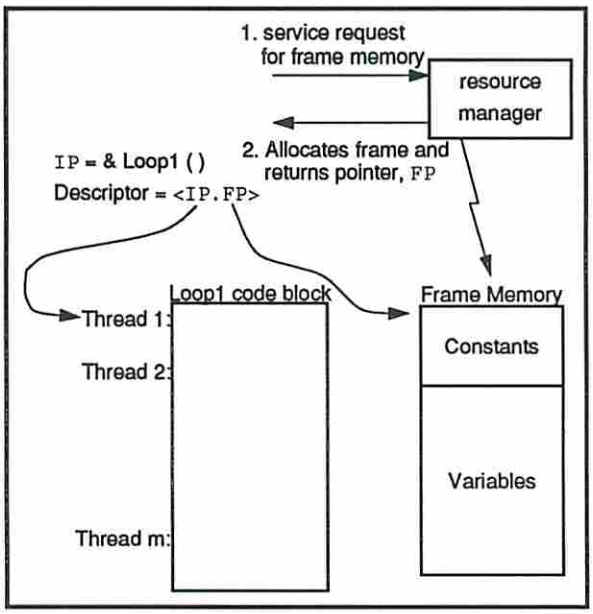

Figure 4.4: An activation is created by allocating the frame memory to store the local environment. The activation is uniquely identified by the descriptor <IP.FP>.

threads (belonging to the current activation) become activated by the arriving data gets longer as latency is increased.

Our motivation was to develop an efficient execution model which reduces the probability of context switches by dynamically generating multiple instances of a given thread that can be computed while the requested remote read operations are in transit. In addition, we want to achieve this without resorting to a hardware-supported multithreaded architecture where the processor switches context at every clock cycle.

Figure 4.4 illustrates the conventional mechanism of creating an activation. In this mechanism, only a single loop iteration instance is active for each activation. Once created, an activation is uniquely identified by a descriptor <IP.FP> where IP is the instruction pointer which points to the next instruction to be executed and FP is the pointer which points to the base of the frame memory. During the lifetime of an activation, multiple threads may be simultaneously ready for execution. However, there can only be a *single* instance of a given thread in an activation. When an activation is executed, its activation descriptor <IP.FP>

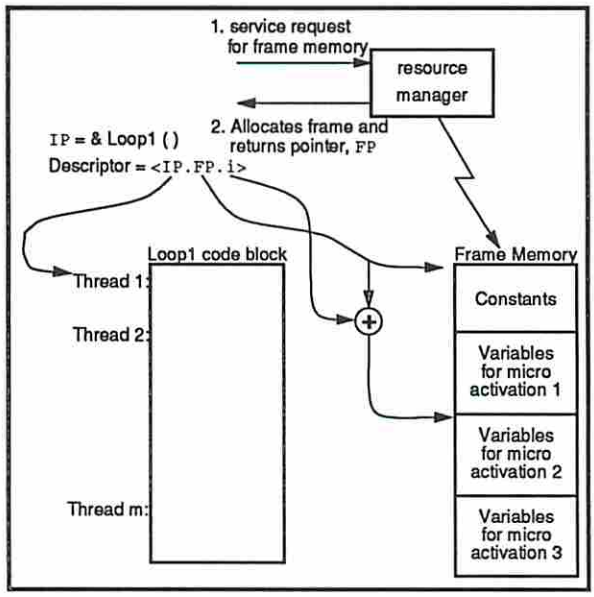

Figure 4.5: In the MIpA model, a macro-activation containing multiple micro-activations is created where the frame memory contains storage for all local environments. The activation descriptor is extended so that different instances of threads can be uniquely identified, <IP.FP.i>.

is stored in the processor registers. Necessary local constants and variables are brought into the processor registers using the FP field of the descriptor.

Our execution model is called the Multiple Iterations per Activation (MIpA) model and it employs a hierarchical activation mechanism in which an activation, once created, further activates multiple instances. Note that this should not be confused with a parent activation which spawns children activations. In our MIpA model, there is no such hierarchical relationship between the spawning and spawned activations. Logically, they are all at the same level. We call the activation containing multiple loop instances a *macro-activation* and each loop iteration instance contained within the macro activation a *micro-activation.* A micro-activation is equivalent to a single conventional activation described in the previous paragraph. In the MIpA model, the process of activating the micro-activation instances requires no additional service from the resource manager. This is because the storage for the local environment of each micro-activation has already been allocated in the activation frame of the macro-activation.

Unlike the conventional method, the MIpA model can have *multiple* instances of a thread existing concurrently within a single activation. This is due to the fact that there exists multiple activations inside the macro-activation in the MIpA model. Thus, if there are **m** threads in a loop body and **c** micro-activations are created within a macro-activation, we have $\mathbf{c} \times \mathbf{m}$ threads in an activation. In the conventional mechanism, **c** is always one. Thus, by grouping **c** loop iterations into a single activation, we have increased the number of potentially ready threads by a factor of **c**. Note that this is exactly what we wanted to achieve, *i.e.*, to have more threads available *within a single activation* that can keep the processor busy while waiting for data from remote locations.

Because multiple instances of a thread can be active concurrently, we need to modify the activation descriptor in order to correctly identify different thread instances within an activation. We have assigned an extra field `i` for this purpose resulting in the following descriptor format: `<IP.FP.i>`. Figure 4.5 illustrates the hierarchical activation mechanism of the MIpA model. For a given micro activation, the corresponding environment within the frame memory is computed using the `FP` in conjunction with the `i` field.

In addition to achieving our main objective, which is to create useful work (threads) that can keep the processor busy while the remote operation is in progress, our MIpA model has other advantages:

1. Since the MIpA model embeds multiple loop instances into a single macro-activation, service calls to the resource manager are less frequent than the conventional method for creating the same amount of work.

2. Because multiple loop activations are encapsulated under one macro-activation, management of multiple activation descriptors is not required although multiple loop activations are, in fact, concurrently resident on a processor.

3. The MIpA model saves memory compared to the conventional scheme because the loop activations that belong to the same macro-activation can share constants from the same area in the frame memory. In the conventional model, because each activation gets separate frame memory, constants need to be copied into each frame memory.

4.3.2 Architectural Support

By introducing a new field `i` into the activation descriptor, regions within the frame memory representing different local environments can be accessed. However, for a processor to compute at its maximum speed, the activation needs to have its environment resident on the processor registers. This poses a problem to our MIpA model when a conventional RISC processor architecture with monolithic register file is assumed.

The problem is that while we can access different regions within the frame memory representing different activations using the `FP` in conjunction with the `i` field, there is no way to distinguish them once they are copied to the processor registers. More specifically, in the MIpA model, multiple loop activations are dynamically created by allocating and assigning separate storage to each activation while the same code block is shared. Therefore, an instruction accessing a set of registers will access the same set of registers regardless of the activation under which the instruction is being executed.

To concurrently support different instances of a thread on the processor registers, we need to be able to distinguish registers which are used to store operands of the same instruction, but different instances. For this purpose, we propose a *Register-Indexed-Register-Access* (RIRA) mechanism which uses the `i` field to access registers belonging to different instances in the similar manner as different regions (environments) within the frame memory are accessed.

A processor providing the RIRA mechanism needs to provide an extra register to store the `i` field in addition to the `IP` and the `FP` registers. These registers are loaded when a new activation becomes resident. During the time an activation is resident on the processor, the content of the `FP` does not change. The contents of the `IP` and the `i` registers change as instructions are executed and the instances of the thread change.

The actual registers that are accessed are the registers which are indexed by the register holding the `i` field of the activation descriptor. An example is shown in Figure 4.6. In the figure, the register file consists of `n` registers where each register can have `m` instances. Therefore, the number of micro-activations that can be encapsulated into a macro-activation at run-time is restricted to `m` which

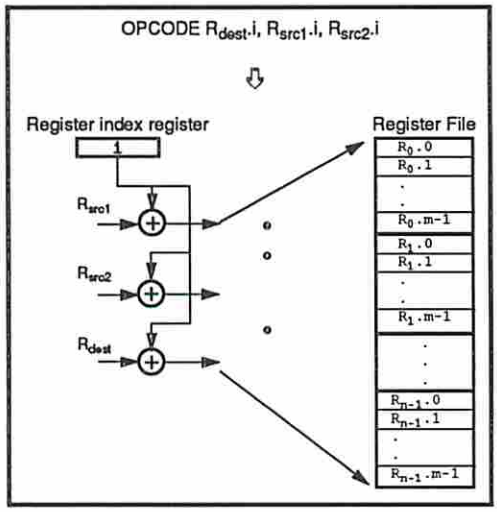

Figure 4.6: The Register-Indexed-Register-Access mechanism is illustrated.

is the maximum number of activations that the processor can concurrently keep resident.

In the figure the registers specified in an instruction are indexed by the `i` register to the register holding the data belonging to activation `i`. Note that the registers does not always have to be used in the partitioned format as required by the MIpA model. If all the registers need to be accessed monolithically for a single instance, the register indexing mechanism can be bypassed.

Figure 4.7 compares the MIpA model with a conventional activation management scheme. Through a mechanism which allows multiple activations to be concurrently managed by a single activation descriptor, the MIpA model, in effect, multiplies the number of threads that can be kept resident on the processor to keep it busy until the requested data finally arrives and trigger more threads. In the conventional case, however, the processor may not be kept busy until the data arrives (due to lack of threads), causing a context switch.

As more registers are implemented to support multiple instances of threads, the context switch overhead will also increase. As a result, while the MIpA model reduces the context switch frequency, the cost of each context switch increases. We can overcome this problem through incremental context switching mechanism which switches context at the micro activation level instead of the macro activation level. To do this, two activation descriptors need to be kept resident on the

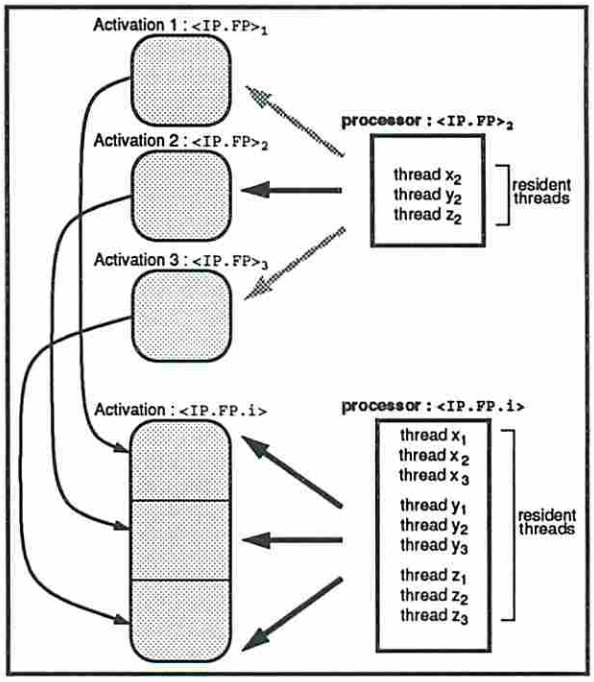

Figure 4.7: The figure illustrates the point that multiple instances of threads can be kept resident on a processor in the MIpA model.

processor. That is, one descriptor for the outgoing activation and the other for the incoming activation. Through this mechanism, only a fraction of the total processor state need to be saved during each context switch.

4.4 Performance Measurements

In this section, we present some preliminary results comparing the performance of the MIpA model and the conventional activation model. The benchmark programs used in the simulation along with the simulation setup is described first followed by a discussion of the performance results.

4.4.1 Benchmark Programs and the Simulation Setup

In our performance measurements, we used two Livermore Loops as benchmark programs. The first benchmark is Loop 1 as shown in section 2 and the second is Loop 7 as shown below.

```
function Loop7 (n:integer; R,T:real; U,Y,Z: OneDim
                returns OneDim)

  for k in 1,n
  returns array of U[k] + R * (Z[k]   + R * Y[k])
          + T * (U[k+3] + R * (U[k+2] + R * U[k+1])
          + T * (U[k+6] + R * (U[k+5] + R * U[k+4])))
  end for
end function
```

The main reason for selecting these two Livermore loops is that they fit the characteristics of the loops we want to address, *i.e.*, the loop body of both loops are quite short with a relatively large number of array read operations. For example, the Livermore Loop 1 has three array read operations and five floating-point operations while Loop 7 has nine array read operations and 16 floating-point operations. Excluding the integer operations used in the array subscript calculations, the array read operations amount to over 35% of the total operations for both loops.

In the performance measurements, the two loops were used as the innermost loop where the loop instances are activated by an outer loop instance which is already active at each processor. In addition, it is assumed that each outer loop instances create inner loop instances only on the same processor. This is illustrated in Figure 4.8.

The following configurations were used in the measurements:

- 64 processors connected in a hypercube network.
- The maximum network latency of 10 clock cycles.
- Each instruction executes in one clock cycle.
- The array elements are randomly distributed across the nodes. No special data distribution/allocation scheme is assumed.
- At the node holding the requested array element, additional memory latency of 10 clock cycles is incurred.

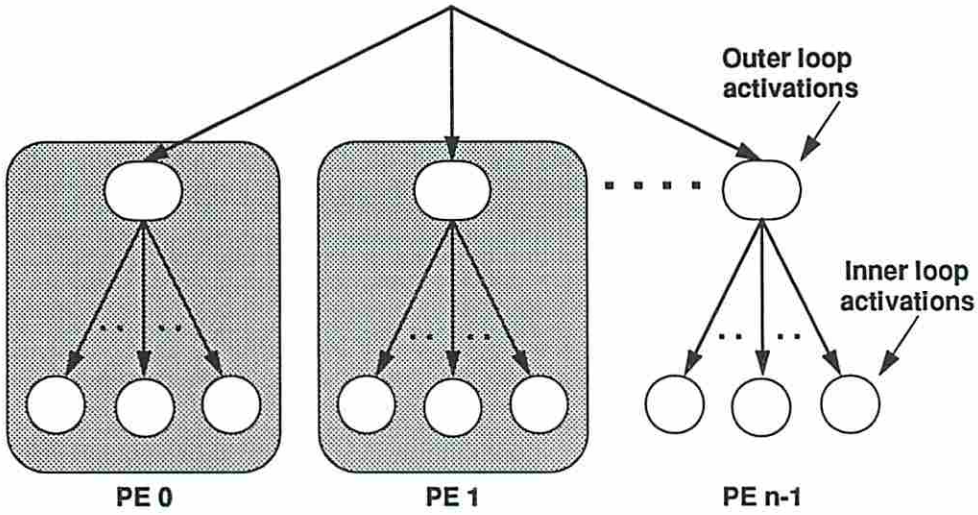

Figure 4.8: The benchmark loops are the inner loops whose activations are spawned on the same processor as the corresponding outer loop activations.

4.4.2 Simulation Results

For each loop, we have measured the execution time of four different configurations by varying the number of activations assigned to each processor from one to six. In the graph shown in Figures 4.9 and 4.10, chunk size refers to the number of activations that can concurrently be supported by the processor state in the MIpA model. Two different configurations are used for the MIpA model. The first configuration fixes the chunk size to two and the second assumes that the processor can support as many as required. The four configurations used in the measurement are as follows:

- No context switch penalty is incurred. The graph marked with circles on Figures 4.9 and 4.10 represent this configuration. This gives the lower bound.
- The conventional activation scheme assuming a context switch penalty of 10 clock cycles for each context switch. The graph marked with rectangles represent this configuration.
- The MIpA model with the chunk size of two. Each context switch incurs a penalty of 10 cycles. The graph marked with triangles represent this configuration.

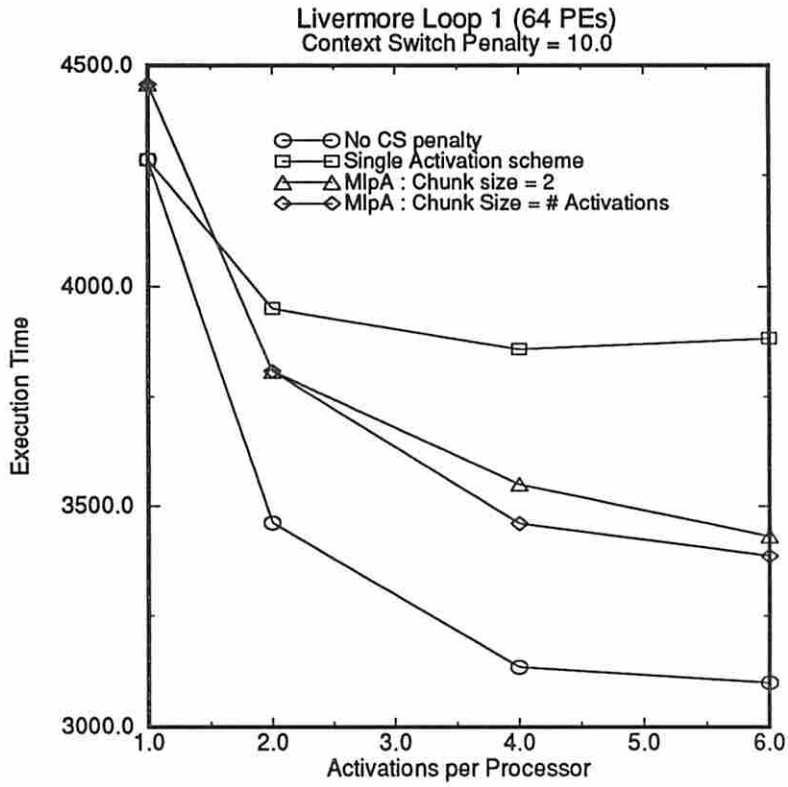

Figure 4.9: The execution times of the Livermore Loop 1 based on four different configurations.

- The MIpA model with the chunk size equal to as many activations as required. Each context switch incurs a penalty of 10 cycles. The graph marked with diamonds represent this configuration.

In Figure 4.9, the execution time for the conventional activation scheme (topmost curve) reaches its minimum when four activations are assigned to a processor. When more activations are assigned, the overhead caused by context switches actually increases the overall execution time. On the other hand, the MIpA model consistently results in a lower execution time as more activations are assigned to a processor. This is in spite of the fact that the MIpA model executes more slowly than the conventional method due to overhead in managing multiple instances within a single activation. This can be seen in the graph when only one activation is assigned to a processor.

Livermore Loop 1, CS Penalty = 10.0				
	Conventional method		MIpA Model (Chunk size=2)	
A/PE	Avg CS/A	Avg CS time	Avg CS/A	Avg. CS time
1	0.00	0.00	0.00	0.00
2	1.02	488.91	0.00	0.00
4	1.58	724.69	0.60	287.97
6	1.78	784.06	0.58	274.69

Table 4.2: The table shows the average number of context switches per activation and the average time spent by a processor for context switches of the Livermore Loop 1: A = Activation, CS = Context Switches

In the MIpA configuration with a chunk size of two (second curve: triangle), the minimum execution time reached is approximately 10% within that of the ideal case where no context switch penalty is incurred. The minimum execution time attained with the conventional scheme is approximately 25% slower than the lower bound. In the case of the Livermore Loop 1, the result suggests that the processor does not need to provide a large number of registers. Between the processor that can support two instances and the one that can support as many as required, the improvement in the execution time is only about 3%. Table 4.2 shows the reason why the MIpA model performs better than the conventional scheme. The average number of context switches per activation for the conventional scheme is greater by a factor of almost three over that of the MIpA model.

In the case of the Livermore Loop 7 (Figure 4.10), it is hardly worth assigning more than one activation to a processor when a conventional method is used. With a slightly improved execution time (approx. 2%) at two activations, the execution time actually increases as more activations are assigned. This is due to slower response times from the array handlers and network nodes which are much more heavily utilized than in the case of Loop 1.

The MIpA model with a chunk size of two performs a little better, but its performance also degrades when four activations are assigned to a processor. In the case of Loop 7, the best execution time is attained when a processor state can support four activations. With this configuration, the execution time is within about 6% of the minimum execution time. The best time reached by the conventional method is slower by approximately 13%. The improvement in execution

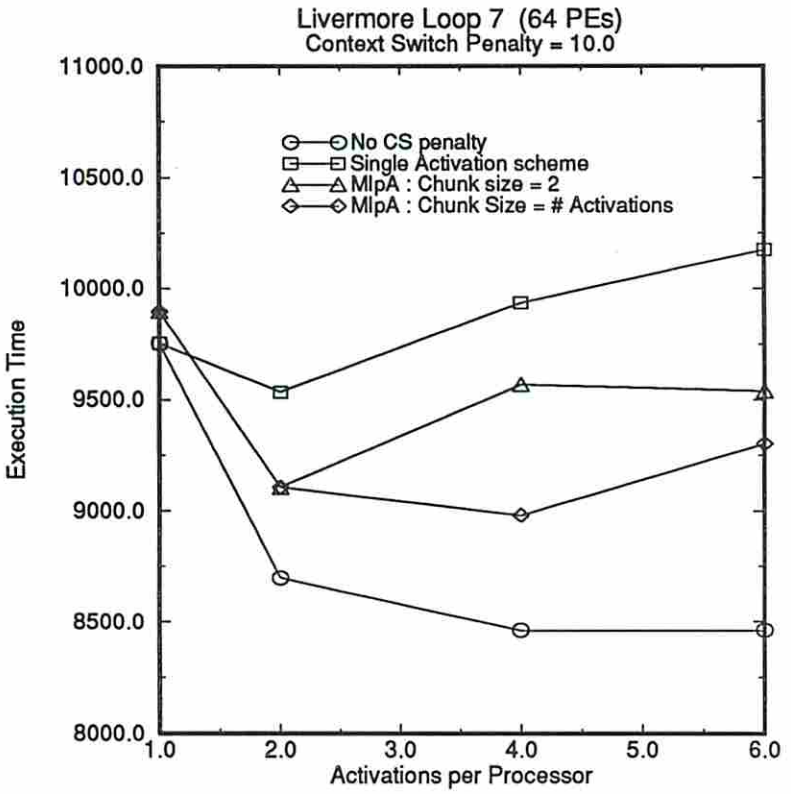

Figure 4.10: The execution times of the Livermore Loop 7 based on four different configurations.

time using the MIpA model over using the conventional method is 50%. Table 4.3 compares the average number of context switches from executing Loop 7 between the conventional scheme and the MIpA model. Although the difference is not as great as in Loop 1, the conventional method still switches context almost twice as often as the MIpA model with the chunk size of two.

4.5 Conclusion

Based on the two benchmark programs, the MIpA model consistently performs better than the conventional method. The reason for its superior performance is due to its execution model which is aimed at reducing the average number of context switches. When compared to a conventional scheme under a same condition, it always resulted in significantly less context switches per activation. Although

Livermore Loop 7, CS Penalty = 10.0				
	Conventional method		MIpA Model (Chunk size=2)	
A/PE	Avg CS/A	Avg CS time	Avg CS/A	Avg. CS time
1	0.00	0.00	0.00	0.00
2	1.74	835.47	0.00	0.00
4	3.21	1473.91	1.35	640.00
6	3.90	1710.94	2.05	880.00

Table 4.3: The table shows the average number of context switches per activation and the average time spent by a processor for context switches of the Livermore Loop 7: A = Activation, CS = Context Switches

more extensive experimentation is needed, the results obtained in this study suggest that a processor supporting relatively few activations may be sufficient to provide a marked improvement over the conventional scheme.

The Livermore Loop 7 showed that for a relatively short loop body with a large number of (possibly remote) array accesses, fine-grain multithreading alone cannot effectively tolerate latency. To obtain a better performance from loops with similar characteristic as Loop 7, latency reduction techniques should be applied in conjunction with the multithreading technique as presented in the paper.

Chapter 5

Array Handling in a Multithreaded Execution Model

One of the important responsibilities of run-time systems for functional languages such as SISAL is efficient memory management. Although the concept of storage does not exist at the language level, the run-time system transparently allocates memory for storing data values when they are created. And once a data value is no longer needed, the run-time system reclaims the memory for later use. Such dynamic memory allocation/deallocation operations are overheads which need to be reduced as much as possible. This chapter presents an array optimization technique which handles arrays without allocating memory at run-time. The content of this chapter is based on a paper published at [65].

5.1 Introduction

Almost all scientific applications that can benefit from parallel computing entail the manipulation of large data structures. Whether the computer is a uniprocessor or a multiprocessor, the latency incurred by memory accesses directly affects performance. Indeed, a well-known solution employed in most uniprocessor systems and shared memory multiprocessor systems is the usage of the cache memory which is aimed at reducing latency by exploiting *locality* (both temporal and spatial).

In large multiprocessor systems where memory is physically distributed across the network, the latency problem is much more severe and unpredictable. This is

because a memory access may be made to a local memory or a remote location where the data must travel through a number of interconnection network nodes. Depending on various factors such as traffic load, interconnection topology, and network bandwidth, the latency can vary over a wide range of values. In such systems, the cache solution by itself is not sufficient.

An alternative solution to the latency problem comes from the data-driven execution model where the problem is indirectly addressed by providing a *latency-tolerant* execution model. The key to this solution is the ability to hide the adverse effects of long latencies by performing useful computations, instead of idling, while the memory access is in progress [27]. In order to effectively overlap computation and communication, the following two requirements must be satisfied:

1. The compiler should be able to expose parallelism of all granularity contained in a program so as to provide a large pool of ready instructions at run-time.

2. The architecture of the target machine should be able to efficiently exploit the parallelism exposed by the compiler.

The following is the current data-flow solution to the above requirements:

1. Use a functional language as a programming language which makes it relatively easy for a compiler to expose existing parallelism.

2. Use multithreading techniques to efficiently exploit fine/medium grain parallelism exposed by the compiler.

3. Use I-Structures [9] so that whenever possible, arrays can be produced and consumed in parallel.

The programming language for our study is SISAL which is a single-assignment functional language [71]. Currently, the Optimizing Sisal Compiler (OSC) is available for a number of shared memory multiprocessor systems (such as Sequent, CRAY, Silicon Graphics, etc.) [20]. Through efficient memory optimizations, the execution times of SISAL programs compiled by OSC compare favorably to those written in FORTRAN on multiprocessor supercomputers such as CRAY-YMP [21].

Although the current implementation of OSC performs well on shared memory systems which utilize a relatively small number of processors, its coarse grain execution model and strict array operations may not be suitable for large multiprocessor systems with distributed memory where latency becomes a crucial performance issue. To better utilize a large number of processors in the system, a finer grain execution model may be desirable. Our strategy is to adopt a multithreading execution model in which a given function body or loop body is compiled into multiple threads. In doing so, a processor can be better utilized by switching to a ready thread whenever a remote memory access is initiated instead of waiting for it to complete.

This chapter presents an optimized array handling technique to be used in conjunction with I-Structures under the new execution model. Currently, I-Structure is the only available scheme which provides fine grain parallelism to the array producer and the consumer. This is achieved by providing non-strict and parallel accesses coupled with *split-phase* memory read operations. The objective is to better utilize processors through the overlapping of computation and communication.

Specifically, each I-Structure memory cell is associated with presence bits which indicate the status of the cell, *i.e.*, *full*, *empty*, or *pending*. Initially, each cell is set to *empty*. If a read request is made to an *empty* cell, the state of the cell is changed to *pending* and the request is queued in the deferred read request list. When the cell is written, the state changes to *full*. If the cell was marked *pending* prior to being written to, all deferred read requests are serviced.

Although I-Structures provide means to better performance in large multiprocessor systems with distributed memory, they must be used with prudence. In the data driven execution model where memory usage is inherently dynamic, frequent allocation/deallocation operations of the I-Structure memory which is physically distributed across the network may create intolerable overhead. In addition, the network may quickly become overloaded due to the three transactions which are required for every transfer of an array element from a producer to a consumer. In the worst case, all three operations may be remote accesses.

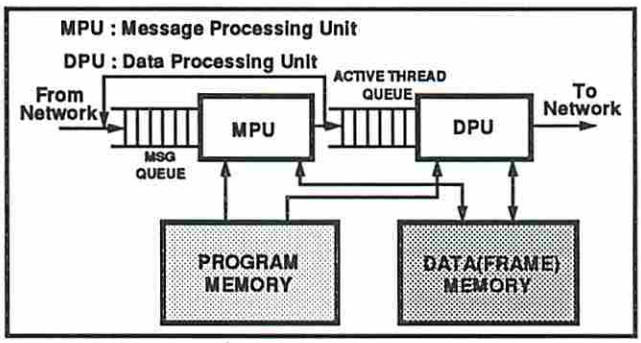

Figure 5.1: The multithreaded architecture model of a processing element.

The objective, therefore, is to propose an optimized array handling scheme called the Direct Array Injection Technique (DAIT) which provides non-strict and parallel array accesses without requiring the expense of allocating storage before use. In addition, by *directly* forwarding the array elements from the producer to the consumer(s), the potential number of remote accesses is reduced compared to that of I-Structures. Although the execution model is quite different, the idea behind DAIT is similar to that of chaining used in vector processors where a computed value from a functional unit is directly fed into the next functional unit instead of first being written into the memory.

5.2 Multithreading Execution Model

The multithreading execution model used for this study is adapted from *T and TAM. A code block corresponding to a loop body or a function body in the program text is divided at compile-time into threads. An instance of a code block is said to be *active* when a frame memory is allocated to it and the input parameters are available. The amount of the required frame memory is determined at compile-time by analyzing the memory requirements of the code block. After a code block has been activated, each thread is executed when its activation requirement is satisfied. A thread executes in a non-preemptive mode, *i.e.*, once a thread has been activated, its execution continues without suspension until all the instructions of the thread are executed.

In general, threads constituting a code block can be classified into the following four classes:

1. **Initialization** thread: There is always one initialization thread in a code block. It is the first thread to be executed in the code block and its function is to receive input parameters and initialize other local constants and variables.

2. **Interface** thread: There can be a number of these threads. Their main function is to communicate with other activations which may or may not be executed on the same processor. For every remote read request, a matching interface thread is required to receive data because of a split-phase operation. An interface thread may update synchronization variables after receiving a token. When the terminal value is reached, the corresponding compute thread is activated.

3. **Compute** thread: These are the threads that actually perform the computations specified in the program as the programmer sees it. A compute thread may be triggered by the initialization thread, interface threads or other compute threads.

4. **Control** thread: This thread determines whether the current activation has done all its required work. For instance, if the code block represents a loop body of a parallel loop, the thread determines whether the frame memory should be reused by another loop iteration or be deallocated.

The multithreaded architecture model used for this study is shown in Figure 5.1. The basic concept behind the model is to decouple the functions of thread activation from that of thread computation. The Message Processing Unit (MPU) is used to execute the interface threads whose main function is to receive tokens and activate compute threads. Once a compute thread is activated, the MPU enqueues the thread descriptor in the Active Thread Queue (ATQ) where it is eventually dequeued by the Data Processing Unit (DPU). The DPU reads a new thread descriptor from the ATQ when the last instruction of the current thread is executed. A thread descriptor consisting of two values, `<IP.FP>`, completely specifies a thread where `IP` is an instruction pointer which points to the

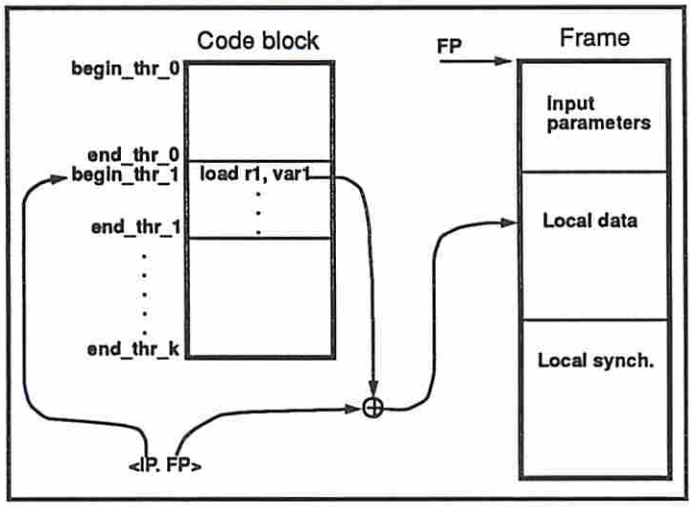

Figure 5.2: The state of a thread is completely specified by a descriptor consisting of <IP.FP>.

first instruction of the thread and FP is a frame pointer which points to the base of the frame memory. Once the IP has been loaded into the IP register of the DPU, subsequent instructions can be fetched and executed by incrementing the register. Memory slots within the frame are accessed using the FP in conjunction with the instruction operands which are offset values. Figure 5.2 shows how the first instruction of a thread is executed using the thread descriptor.

5.3 The Direct Array Injection Technique

We will now describe the Token Relabeling method which inspired the development of our array handling scheme called the Direct Array Injection Technique before describing the implementation of the latter in a multithreading context.

5.3.1 The Token Relabeling Method

The Token Relabeling (TR) method has been proposed for handling arrays in the tagged token data-flow architecture [43]. The central point of this scheme is that, unlike other methods, the array elements should not be stored in the structure store facilities. Instead, each array element is treated as a scalar value and is carried in a token. Individual array elements are uniquely identified by the

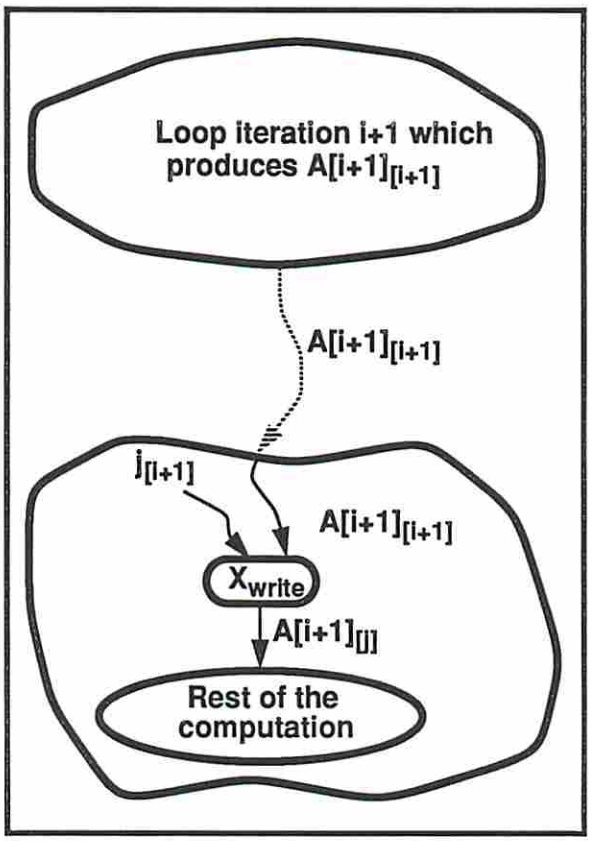

Figure 5.3: A token carrying an array element needs to be relabeled appropriately in order for it to be forwarded to the correct consumer context.

iteration tag field. The basic array accessing mechanism of the TR method is also different from the other array handling methods using a structure store.

In the I-Structure scheme, for example, array elements are first sent to the I-Structure handler to be written into the corresponding memory location. Likewise, an array consumer sends a read request to the I-Structure handler which services the request when the data becomes available. In the Token Relabeling method, each array element is sent directly from the producer to the consumer, bypassing the structure store. Array elements are temporarily stored in the waiting storage pool of the consumer's match unit until they consumed. Functionally, the Token Relabeling method is identical to the I-Structure approach. Both schemes provide non-strict array accesses and deferred reads. The difference is that the Token Relabeling method does it without special structure storage.

A common scenario for applying the Token Relabeling method is a situation where parallel loops form a producer-consumer relationship such that one or more arrays produced by the producer loop are consumed by one or more loops. The basic token relabeling rule is that an array element $A[i]$ is computed by the producer loop iteration i and is tagged by i, $A[i]_{\{i\}}$. Likewise, a consumer loop iteration j consumes array element $A[k]_{\{j\}}$. If $A[i]_{\{i\}}$ is consumed by the loop iteration $j = i$, no token relabeling is required. However, if $A[i]_{\{i\}}$ is consumed by the consumer loop iteration $j \neq i$, array element $A[i]$ must be appropriately relabeled for correct consumption. For example, assume a consumer iteration j consumes $A[i+1], j \neq i+1$. Then $A[i+1]_{\{i+1\}}$ produced at iteration $i+1$ must be relabeled to $A[i+1]_{\{j\}}$ so that the token is correctly consumed by the consumer iteration j. Figure 5.3 shows how the iteration field of the tag is appropriately relabeled before the array element is consumed.

The advantages of the Token Relabeling method over I-Structures are that, 1) no special resources are required to provide parallel non-strict array accesses and that 2) communication is reduced. The Token Relabeling method is a more efficient technique because, for every array element transfer from a producer to a consumer, at most only one remote access is required. On the other hand, there are potentially three remote transactions when I-Structures are used. A study has shown that given the same program, the TR method generates less network traffic than I-Structures resulting in faster execution time [46]. The Token Relabeling method has been also applied to a scheme which exploits parallelism at run-time that was hidden at compile-time [62]. However, there are a number of drawbacks to using the Token Relabeling method. One of the drawbacks is that expensive match unit storage space is taken by the array elements. Therefore, the TR method is best suited for those situations in which the array in question behaves like a temporary variable, *i.e.*, when the array is produced and consumed relatively quickly. On the other hand, an I-Structure would be more appropriate when the array behaves like a global variable.

5.3.2 DAIT in a Multithreading Execution Model

The Direct Array Injection Technique in a multithreading execution model consists of the following compile-time and run-time features:

- A compile-time analysis which determines the loops applicable for DAIT.
- A compile-time analysis which determines the individual producer and consumer loop iterations that are "connected" at run-time to transfer array elements.
- A new run-time technique which allows multiple loop instances to be active simultaneously using the same frame memory.
- A communications protocol which reduces the network traffic by reducing the number of messages required to transfer a given number of array elements.

The first step toward DAIT is the identification of the applicable sets of loops. A graph analyzer has been implemented which scans the program graph for loops appropriate for DAIT. The program graph is a data dependency graph called the Intermediate Form 2 (IF2) [96] which is the optimized Intermediate Form 1 (IF1) graph [88]. The IF1 graph is produced by the SISAL frontend. Our current implementation picks producer-consumer pairs for DAIT according to a criterion based on the following definitions. The ForAll loop used in the definitions refers to the SISAL parallel loop construct.

Def. 1 : A ForAll loop is a **candidate producer (consumer)** if it produces (consumes) arrays which are consumed (produced) by one or more ForAll loops.

Def. 2 : A pair consisting of a candidate producer and a candidate consumer forms a **simple P-C set** if the candidate consumer consumes arrays produced by the candidate producer and the candidate producer is the only (array producing) immediate parent of the candidate consumer.

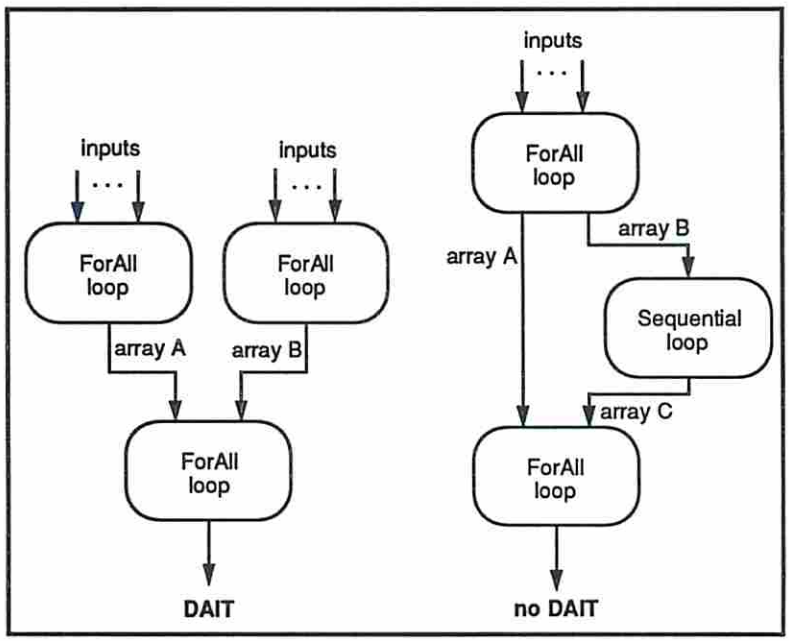

Figure 5.4: The set of three ForAll loops on the left of the figure satisfies the criterion for DAIT while the ones on the right do not.

The graph analyzer has been tested on a multigrid [17] PDE solver which uses the Red-Black Gauss-Seidel iteration to compute the grid points [79]. This is a widely used parallel version of the basic Gauss-Seidel iteration which lacks parallelism, but provides a good convergence rate. The Red-Black Gauss-Seidel iteration consists of two parallel loops in sequence. The first loop computes the red grid points where the resulting array is used by the second loop to compute the remaining black grid points. Our graph analyzer picked out these two loops along with several other loops in the multigrid solver to be applicable for DAIT which all happen to belong to the multigrid solver's kernel. Although it is difficult to predict the applicability of DAIT based on a single experiment, preliminary results point to a wide applicability of the technique, at least, to many iterative PDE solvers.

The next step, after the selection of the producer-consumer sets, is to determine which consumer loop iteration j consumes an array element produced by the producer loop iteration i. This step corresponds to the token relabeling operations in the data-flow architecture which was done at run-time. However, if the array access pattern of the consumer loop can be determined at compile-time, the run-time overhead can be avoided by statically performing the relabeling. In fact,

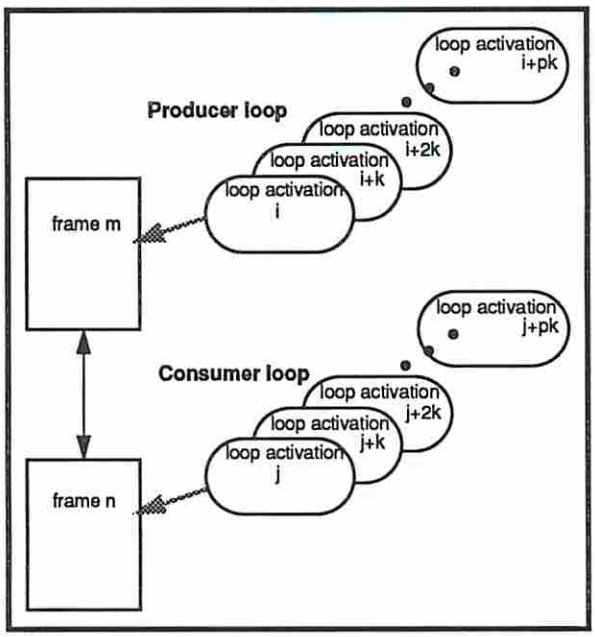

Figure 5.5: Because frame memory is reused, some synchronization is required to make sure that the correct producer-consumer loop instances are active before the data is transferred.

the current version of the direct access scheme works only when the array access pattern can be determined at compile-time. Fortunately, many PDE solvers based on the finite difference method or the finite element method result in a regular banded sparse matrix which can be handled by the current DAIT technique.

In the current multithreading execution model, a frame is used by a single active instance of a code block. In this scheme, `<IP.FP>` completely describes the instance of the code block. This approach, however, can result in low processor utilization when the Direct Array Injection Technique is used. This is caused by the combination of two facts. First, the fundamental idea of DAIT is that no storage space is allocated for an array. Second, in most real world situations, loops are too large to be completely unraveled due to limited memory resources. Consequently, only a limited number of loop iterations can be active simultaneously and the frame memory of a terminated loop instance is reused by a new active loop instance.

The consequence of these two conditions is that whenever an active producer loop iteration is ready to send an array element to its consumer, it must first

make sure that the correct consumer loop iteration is also currently active. In other words, although the mapping between a producer and a consumer frame memory is static, the loop instances which become active on these frame memory blocks change dynamically (Figure 5.5). In the figure, it is assumed that a frame used by a loop iteration `i` is reused next by iteration `i+k` where `k` is the maximum number of loop activations that are allowed to be active concurrently. Since the duration of an active period of a loop instance is different between a producer and a consumer, a synchronization is necessary every time an array element is ready for transmission. Since there may not be any useful computation (within the activation) to perform during the synchronization phase, a context switch to another active instance may be necessary. Assuming that context switches among different activations are much more expensive than the context switches among threads within an activation [27], this may result in poor processor utilization.

To overcome this problem, we have developed the Multiple Iterations per Activation (MIpA) technique which, unlike the conventional approach (call it the Single Iteration per Activation (SIpA)), allows multiple loop instances to be *simultaneously* active on the same frame memory. Because there can be multiple instances of the same thread simultaneously, a new descriptor is required to distinguish each thread instance. A new field `i` is added to the descriptor `<IP.FP.i>`. The `i` field can be viewed as a micro descriptor within a code block to distinguish different thread instances. In this new scheme, there is a higher probability of useful computation during the synchronization phase thereby increasing processor utilization.

To be used in conjunction with the MIpA technique, a new communication protocol is introduced which can further increase performance by reducing the network load. This new communication protocol takes advantage of the fact that there are `w` slots available to store array elements in the consumer frame memory (where `w` is the number of iterations that are simultaneously active on that frame memory). Therefore, instead of performing synchronization at every iteration, a single synchronization point is sufficient for every `w` iterations (Figure 5.6). By using this communication protocol concurrently with the MIpA technique, we have the following advantages:

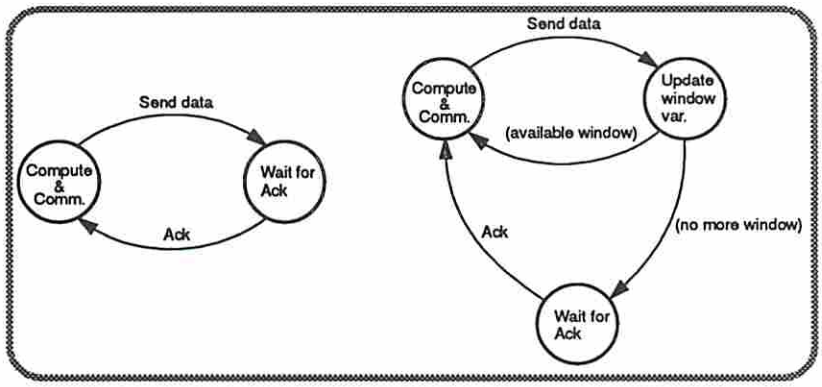

Figure 5.6: Simplified state transition diagram of the communications protocol.

1. The number of calls to the resource manager is decreased by lumping multiple instances into a single frame without loss of parallelism.

2. The synchronization between the producer and the consumer instances is not required at every iteration. As a direct consequence, the processor is better utilized while the network traffic is reduced.

The top diagram in Figure 5.7 shows that when the conventional SIpA scheme is used, synchronization is required every time a new loop instance becomes active. The bottom diagram shows that when the MIpA scheme is used with the windowing protocol, a processor can be better utilized while reducing the network load. The variable `w` in the bottom diagram of Figure 5.7 indicates the number of simultaneously active instances using the the same frame. Let us now examine the processor utilization of each scheme over m iterations. The total execution time of an activation is $T_{comp} + T_{comm}$ where T_{comp} is the computation time and T_{comm} is the communication time. Then for the SIpA case,

$$SIpA_{proc\_util} = \frac{mT_{comp}}{m(T_{comp} + T_{comm})} = \frac{T_{comp}}{T_{comp} + T_{comm}}$$

In the MIpA case, assume that w iterations can simultaneously be active on a frame. We then have the following expression,

$$MIpA_{proc\_util} = \frac{\lceil \frac{m}{w} \rceil wT_{comp}}{\lceil \frac{m}{w} \rceil (wT_{comp} + T_{comm})} = \frac{T_{comp}}{T_{comp} + \frac{T_{comm}}{w}}$$

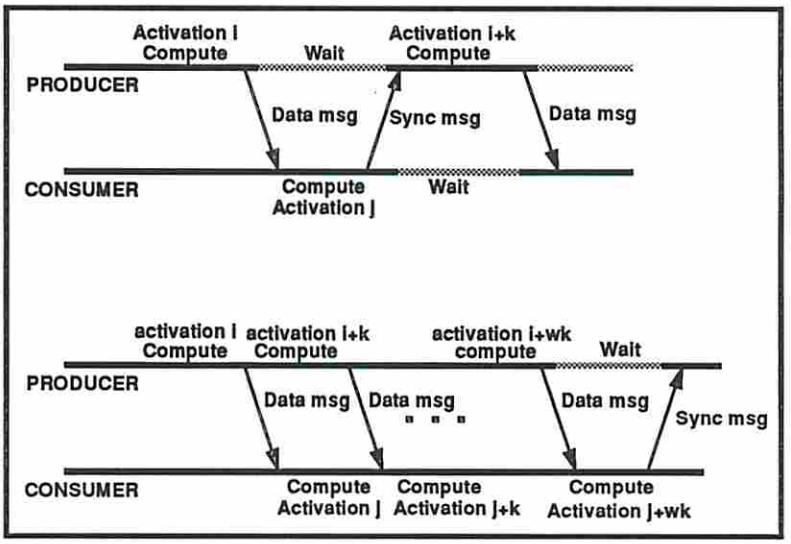

Figure 5.7: The upper figure shows that synchronization is required at every iteration when conventional activation mechanism is used. The lower figure shows that a processor is better utilized when multiple instances are lumped into a single frame.

5.4 Performance Measurement

This section reports on the performance of three different implementations using the one-dimensional version of the popular Red-Black Gauss-Seidel iteration program as a benchmark. Each version is implemented using a different array handling technique and its resulting performance is discussed.

5.4.1 The Simulated Architecture

The simulated architecture of a processing node used in the performance measurement is as shown in Figure 5.1. Each node consists of a Message Processing Unit (MPU) and a Data Processing Unit (DPU) which operate asynchronously to each other. The main function of the MPU is to handle incoming messages and if a thread is activated as a consequence, its descriptor is inserted into the Active Thread Queue (ATQ) for eventual execution by the DPU. As thread scheduling is not the main research issue here, a simple FIFO policy was adopted. The execution of an instruction is assigned one time unit assuming that one instruction

can be issued at every clock cycle. A hypercube topology was selected for the interconnection network.

In our simulator, the I-Structure handler is implemented with the following key implementation parameters:

- Array elements are low-order interleaved across I-Structure (IS) nodes; an array element $A[i]$ is mapped to the IS node, $inode = i \ \ Mod \ \ M$, where M is the total number of I-Structure nodes in the system. There are as many IS nodes as there are processing nodes.
- The cost of the I-Structure memory allocation and the initialization time is idealized to zero.
- Within the IS node, each memory read and write operation takes one time unit. Every enqueue and dequeue operation to/from the deferred read queue also takes one time unit.
- No extra cost is assigned for accessing an IS node.
- The processing node in which an array producer activation executes and the IS node to which the produced array element is stored are allocated randomly. Therefore, i-write or i-fetch operations are not guaranteed to be always local.

5.4.2 Benchmark and Conditions of Experiments

Three versions of a one-dimensional Red-Black Gauss-Seidel iteration method have been implemented to observe the effects of different array handling techniques on the overall performance of the program. A SISAL version of the program is shown in Figure 5.8. One of the three versions is implemented using I-Structures with the assumptions listed in the previous subsection. The other implementations are the two versions of the Direct Array Injection Technique. In the first version, a single iteration is active at a time in a given frame memory and the frame is reused only after the current activation is terminated. This implementation is called the

```
type OneDim = array[double_real];

function rbGS(N:integer; Omega:double_real; A:OneDim returns OneDim)
    let
        h := 1.0d0/double_real(N);
        TempRow1 := for i in 1,N-1
            InnerValues := if mod(i,2) = 1 then
                let
                    RealI := double_real(i);
                    Fi,Ki,K1,K2 := ComputeF(h,RealI);
                in
                    (K1*A[i-1] + K2*A[i+1] + Fi) / Ki
                end let
            else
                A[i]
            end if
        returns array of InnerValues
        end for;
        FirstRow := array_addh(array_addl(TempRow1,A[0]),A[N]);
        TempRow2 := for i in 1,N-1
            InnerValues := if mod(i,2) = 0 then
                let
                    RealI := double_real(i);
                    Fi,Ki,K1,K2 := ComputeF(h,RealI);
                in
                    (K1*FirstRow[i-1] + K2*FirstRow[i+1] + Fi) / Ki
                end let
            else
                FirstRow[i]
            end if
        returns array of InnerValues
        end for;
    in
        array_addh(array_addl(TempRow2,A[0]),A[N])
    end let
end function
```

Figure 5.8: Sisal program of a one dimensional Red-Black Gauss-Seidel iterative method.

DAIT-Single Iteration Frame (DAIT-SIpA). In the second version, multiple iterations are active simultaneously in a given frame memory. This implementation is called the DAIT-Multiple Iteration Frame (DAIT-MIpA). All three implementations are hand coded for the simulation.

Two different performance measurements were carried out for each implementation under various network latency conditions. For the speedup measurement, the grid size of the program was fixed at 2000 and the number of processors was scaled from 1 to 64. For each implementation, the execution time was measured at two different latencies (1 and 10). The second measurement is the data scalability inspired by [51]. Ideally, for a given program with enough parallelism, the execution time should stay constant if the problem size and the number of processors are increased simultaneously by the same factor. In reality, the execution

Red-Black Gauss-Seidel iteration, Data size = 2000						
	DAIT-MIpA		**I-Structures**		**DAIT-SIpA**	
PE	L=1	L=10	L=1	L=10	L=1	L=10
1	108083	108083	104155	104155	76455	76455
2	53982	53978	52109	52139	38392	38483
4	27137	27265	27045	30700	19384	24976
8	14599	16329	13076	20908	9997	19021
16	7970	11755	7188	16594	5511	14000
32	4597	8230	3469	10274	3758	10090
64	2623	5283	2187	8434	3396	7091

Table 5.1: Execution time of each implementation when the problem size is fixed at 2000 and the processors are scaled.

time will change depending on various factors arising from the compile-time and run-time characteristics as well as the machine architecture.

To measure the data scalability, the data size was varied from 500 to 4000 and the number of processors from 16 to 128. Since the actual amount of parallelism is capped by a loop-bounding mechanism, the maximum number of activations allowed concurrently was also scaled accordingly from 96 to 768; on the average, 6 iterations are active concurrently at each processing node. The measurements were repeated for four different communication latencies (L = 0, 1, 5, and 10 time units). We define the scaling factor SF as the ratio between the execution time of the reference set and the execution time when the processors and the data size are scaled:

$$SF(PE, DATA\ \ SIZE) = \frac{EX_{ref}}{EX(PE, DATA\ \ SIZE)}$$

The EX_{ref} is the execution time of the smallest set (the reference set), *i.e.*, $EX(16, 500)$ for each latency condition.

5.4.3 Observations

Table 5.1 lists the execution times of each implementation as the processors are scaled from 1 to 64 while the problem size is fixed at 2000. Among the three implementations, the DAIT-MIpA performs best when a larger number of processors

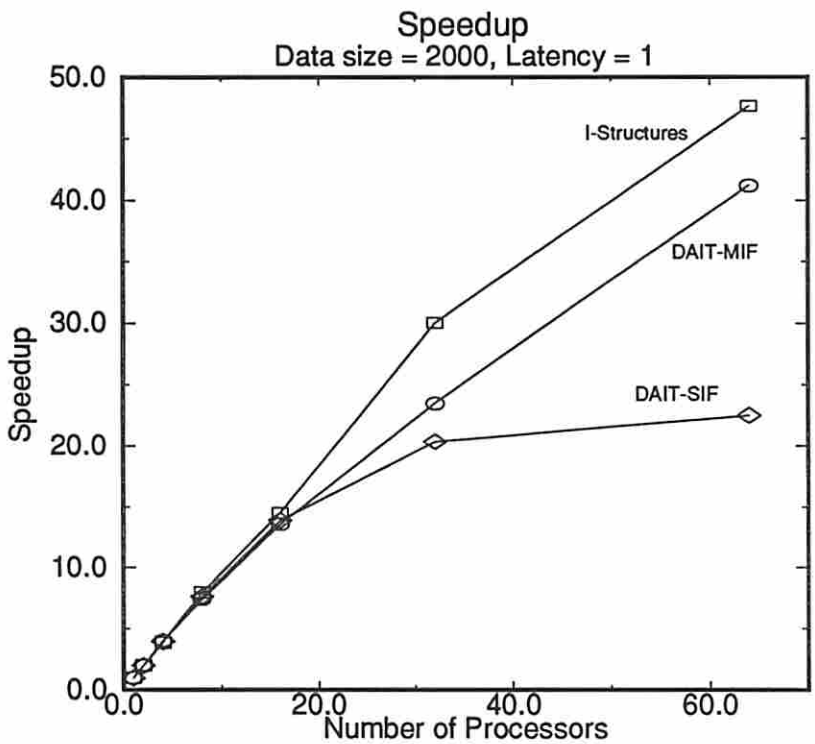

Figure 5.9: The speedup of three different implementations when the network latency is 1.0.

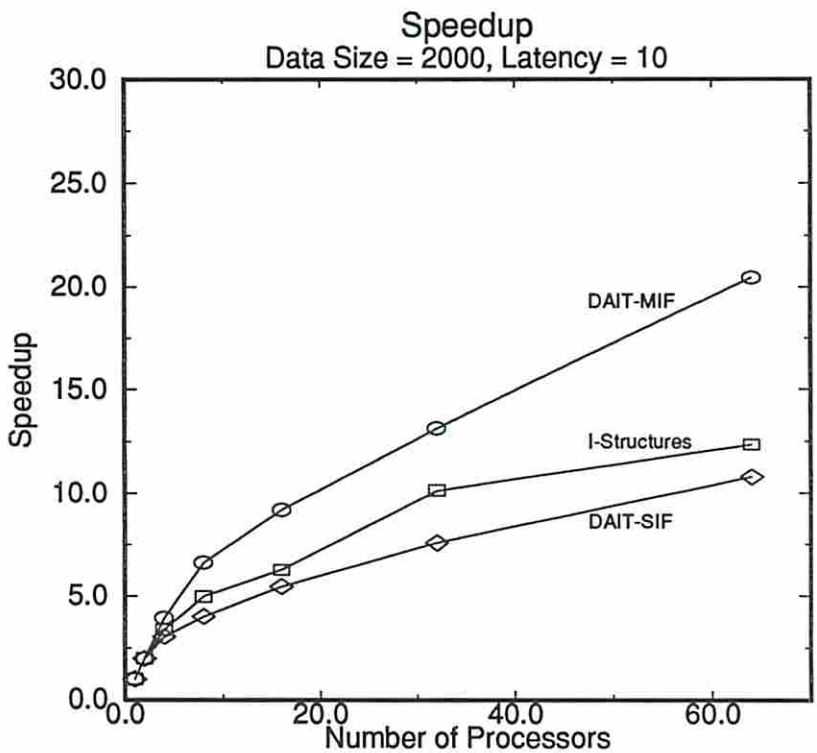

Figure 5.10: The speedup of three different implementations when the network latency is 10.0.

are used under longer latency condition. With the latency set to 10, the DAIT-MIpA produced the fastest execution time for 8 or more processors. For latency of 1, the I-Structure version had the fastest execution time when the number of processors were 32 and 64. For a smaller number of processors, the DAIT-SIpA implementation resulted in the best execution time. Figure 5.9 shows the speedup curve of each implementation at latency = 1. Under this condition, the I-Structure implementation has a better speedup ($\approx$ 47.5). Because of the inefficient frame usage, the speedup curve of the DAIT-SIpA implementation quickly saturates after 32 processors. When the latency is increased to 10, however, the DAIT-MIpA produces the best speedup ($\approx$ 20.5) as shown in Figure 5.10. At this latency condition, the speedup of the I-Structure version is reduced to 25 % of the speedup at latency = 1 while the speedup of the DAIT-MIpA and the DAIT-SIpA is only reduced by half.

Table 5.2 shows the execution times while Figures 5.11 and 5.12 show the corresponding data scalability curves of each implementation under different latency conditions. Each point in Figures 5.11 and 5.12 is computed according to the scaling factor formula. For example, the scaling factor $SF(128, 4000)$ of the DAIT-MIpA at latency = 10 is computed by $EX(16, 500)/EX(128, 4000) = 0.80$. To avoid cluttering, only the data scalability curve for the DAIT-MIpA and the I-Structure implementations are shown. In terms of absolute execution time, the I-Structure implementation performs the best for small latencies (L=0,1). On the other hand, DAIT-MIpA produces the best result for longer (more realistic) latencies (L=5,10) and larger processors ($\geq$ 32). In terms of data scalability, DAIT-MIpA consistently performs better for all latencies. As can be seen from the graphs, the scalability of the DAIT-MIpA degrades to 0.8 (L=10) and to 0.67 (L=5); the scalability of the I-Structure implementation degrades to around 0.47 for both latencies. For a realistic latency value, DAIT-MIpA performs better in both the execution time and data scalability. On the other hand, the DAIT-SIpA performs poorly against the other two implementations. Its only best execution time is for two cases when 16 processors (L=5,10) are used.

DAIT-MIpA				
Latency	PE:Data Size			
	16:500	32:1000	64:2000	128:4000
0	2794	2687	3195	4111
1	2788	2711	3218	4176
5	3107	3217	3799	4678
10	5604	4790	5786	6995
I-Structures				
Latency	PE:Data Size			
	16:500	32:1000	64:2000	128:4000
0	1742	1844	2121	3756
1	1742	1848	2168	3792
5	2691	3755	4751	5785
10	5271	7506	9330	11151
DAIT-SIpA				
Latency	PE:Data Size			
	16:500	32:1000	64:2000	128:4000
0	1935	2668	4603	8743
1	1953	2699	4624	8794
5	2483	3458	5043	9190
10	4483	6009	7091	10012

Table 5.2: Execution time of each implementation when the data and the processors are scaled.

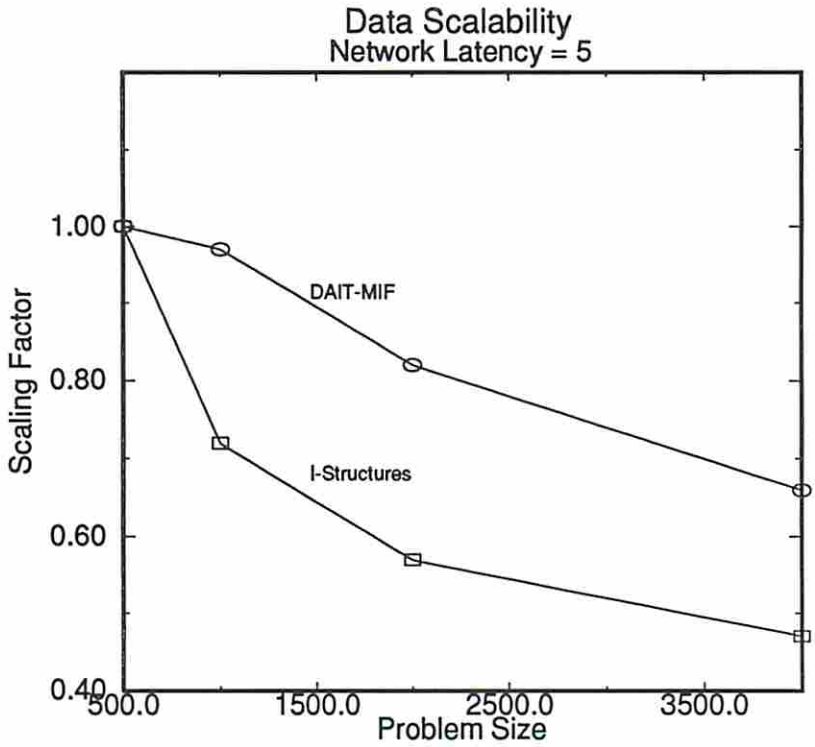

Figure 5.11: The comparison of data scalability when the network latency is 5.0.

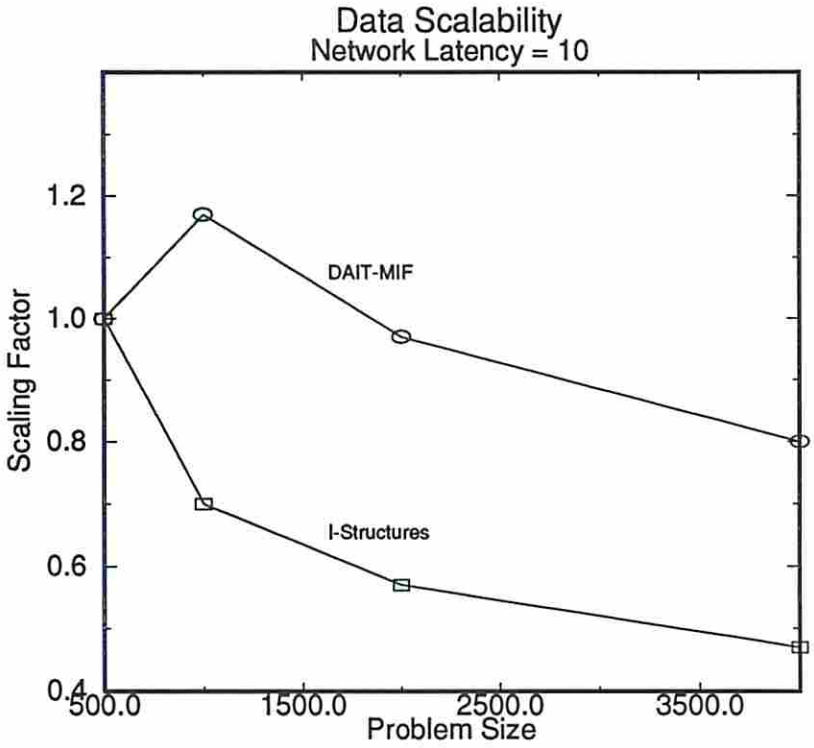

Figure 5.12: The comparison of data scalability when the network latency is 10.0.

5.4.4 Interpretation

The key to achieving good performance is to effectively overlap computation and communication. At the same time, available resources must be utilized efficiently. In both the speedup and the data scalability measurements, we have observed the performance of the I-Structure version degrading rapidly for longer latency conditions. The cause of such performance characteristics is due to the over-utilization of the network by the I-Structure implementation. Although split-phased remote memory operations provide an opportunity for efficient processor utilization, network resources may easily be overloaded with remote message traffic and become a bottleneck.

In the one-dimensional Red-Black Gauss-Seidel program, each producer loop iteration creates one array element and each consumer loop iteration consumes two array elements.. Assuming the array size is N, the worst case expression for the total number of remote array operations is:

$$Total\ remote\ msgs = i\text{-}write\ msgs + i\text{-}fetch\ msgs + data\ msgs = 5N$$

In the DAIT-SIpA implementation, the network is better utilized than in the I-Structure implementation because the message traffic is reduced. This is achieved by the direct array accessing mechanism in which the array elements are sent directly from the producer to the consumer activations; write operation is always a fast local memory write operation to a frame memory. The expression for the total number of remote messages is:

$$Total\ remote\ msgs = data\ msgs + sync\ msgs = 4N$$

On the other hand, this scheme potentially under-utilizes processors because of the required synchronizations between the producer and the consumer activations.

The DAIT-MIpA mechanism solves the low processor utilization problem by allowing multiple iterations to be active simultaneously on a single frame memory block. This also reduces remote message traffic since a synchronization is not needed at every array element transfer. The total number of remote messages

for the DAIT-MIpA is as follows (where w is the number of iterations that are allowed to be active simultaneously on a frame memory block):

$$Total\ \ remote\ \ msgs = data\ msgs + sync\ msgs = 2N + 2\lceil\frac{N}{w}\rceil$$

From the above expressions for the total number of remote messages, we expect that the I-Structure implementation would cause most network traffic. Figures 5.13 and 5.14 show the processor and the network utilization of the three implementations of the Red-Black Gauss-Seidel program at different latency values when the *same number of activations* are allowed to be active simultaneously. For the reasons discussed, the I-Structure implementation saturates the network for latencies of 5 and above, although it performs well under low latency conditions. In fact, since no synchronization is required, I-Structures would always perform better than the other two implementations in an ideal condition (no network latency).

For the I-Structure implementation, it is difficult to increase processor utilization under longer latency conditions because the network rapidly becomes saturated. In the DAIT-SIpA implementation, processors are consistently underutilized. In the DAIT-MIpA implementation, the network utilization is still around 50 % even at latency=10 making it possible to activate more loop iterations. Among the three implementations, the DAIT-MIpA implementation utilizes the processor and the network resources most efficiently given the same conditions.

5.5 Conclusions

We have described in this chapter a new array handling mechanism called the Direct Array Injection Technique, measured its performance, and compared it with that of the I-Structure representation on a simple parallel loop. Through mechanisms that better utilize processors and reduce network traffic, it resulted in a superior performance under more realistic latency conditions. Programs can be compiled with this array handling scheme for better performance.

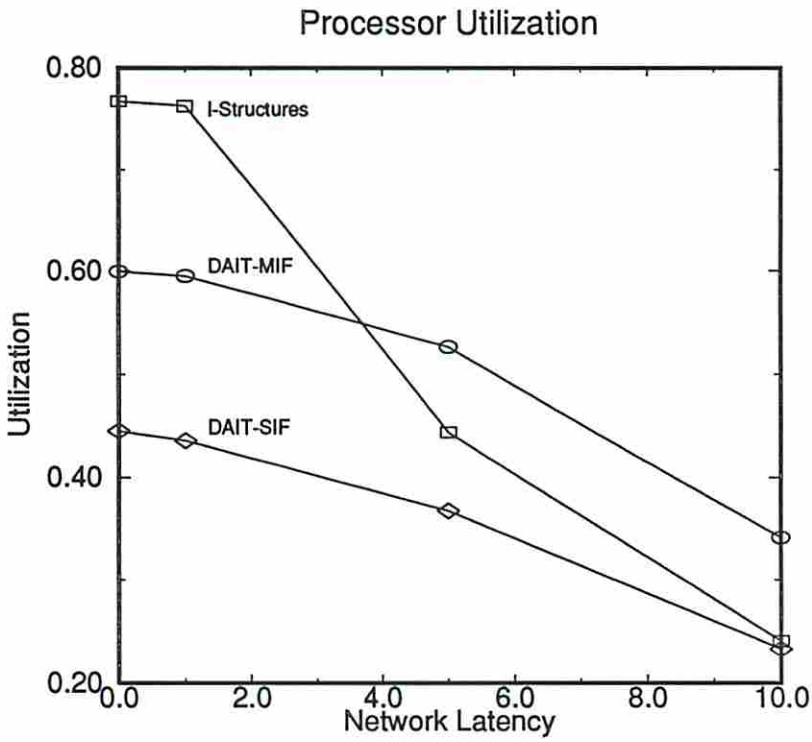

Figure 5.13: Processor utilization of the three implementations for different latency conditions.

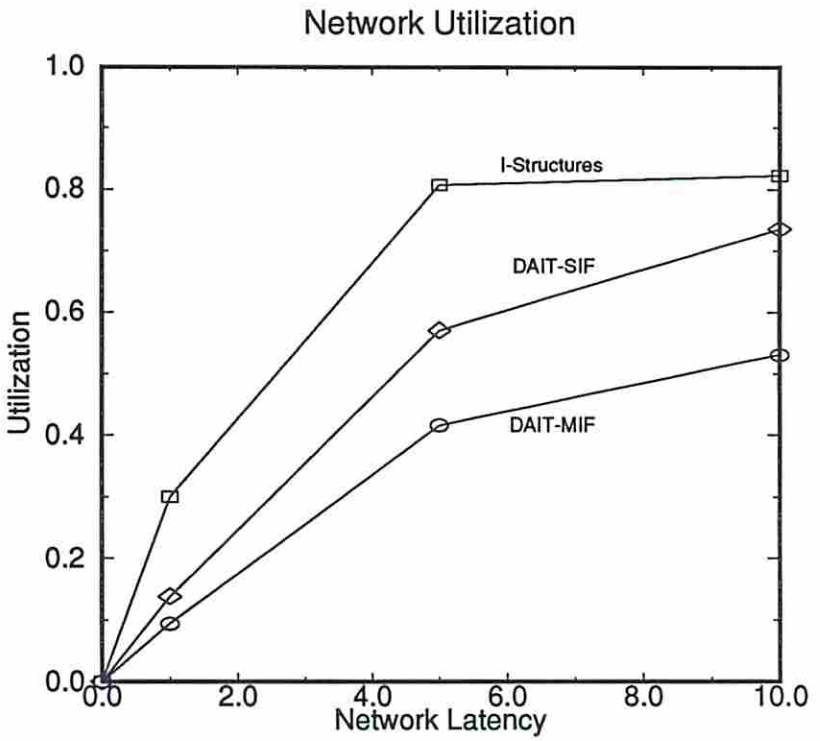

Figure 5.14: Network utilization of the three implementations for different latency conditions.

The Direct Array Injection Technique, however, is not meant to be used at the exclusion of other array handling methods. For those arrays which are accessed over a long period of time after being produced, the I-Structure approach is still the most appropriate array handling method which provides fine grain parallelism. The objective is to use the Direct Array Injection Technique selectively in conjunction with the I-Structures to enhance run-time performance. A preliminary examination on a multigrid iterative PDE solver using a graph analyzer shows that the Direct Array Injection Technique can be applied to the portions corresponding to the kernel of the program. Since most of the execution time is spent on these parts, DAIT can significantly improve the performance of the program.

Chapter 6

Dynamic Parallelism

In many real world scientific applications, there are situations where a compiler is not able to determine whether a loop can be executed in parallel or not because the array subscripts are in such a format which makes the compiler unable to determine data dependency relationship. For example, when a coefficient matrix from some discretization process such as finite element method is assembled at run-time, the array subscripts are stored as arrays. Although there may be a significant degree of parallelism, the loop is executed sequentially. This chapter discusses a technique which exploits such run-time parallelism in a data-flow framework. The presentation is based on the paper published in [62].

6.1 Introduction

The data-flow model of computation is an alternative to the von Neumann model of computation in that it allows maximum parallelism as provided by a given program or algorithm. Nodes in data-flow graphs represent instructions. A node is a single instruction in the case of a micro data-flow and may represent a set of instructions in the macro data-flow model. Edges in data-flow graphs correspond to a data dependency between the nodes connected. Thus if there is an edge from node A to node B, node B can only be executed after node A has been executed. Nodes which are not connected may be independent of each other and thus may be executed in any order or even in parallel. The firing or execution of a node is dependent only on the availability of its input operands [7, 5].

Structure handling is one of the main issues in the implementation of the data-flow model of computation. This is due to one of the key data-flow principle which demands computing to be free from side-effect or in other words, to be fully functional. This desirable characteristic forces careful consideration in deciding what kind of structure handling schemes should be used. Token Relabeling is a technique which was proposed for handling data structures such as arrays in data-flow computers [43, 46]. Unlike other structure handling schemes [9], the Token Relabeling method does not require an explicit structure store. In Token Relabeling, each array element is treated just like any other scalar tokens and is uniquely identified by the iteration field of the tag. Thus the production and the consumption of array elements does not require array handling actors such as APPEND and SELECT to store and read the array elements using some dedicated structure store (this would be the case for other structure handling schemes with an explicit structure store).

Figure 6.1 contrasts the data-flow graphs for some array operations with an explicit structure store and with the Token Relabeling method. When Token Relabeling is used, an array element is directly sent to its consumer once it is produced and tagged[1] without being stored in a structure store. A selection process is simply done by matching the desired tag values, *i.e.*, the iteration field of the tag holds the desired index value. Because of the direct forwarding of array elements by the producer to the consumer, Token Relabeling is also called a direct access approach while that of I-Structure is an indirect access approach [46, 95].

It has been noted that a large part of scientific computing deals with sparse matrices of irregular structure [81]. Such matrices are produced, for example, from the finite volume modeling of a complex object such as the inside of a jet engine using unstructured grids instead of structured rectangular grids in computational fluid dynamics applications. In such applications where access of array elements is irregular, it is common for an array to be subscripted by another array. Such indirect addressing[2] is one of the main reasons why sparse matrix computations

[1] It is not always required to re-tag an array element after it is produced. Tagging is required only if the array subscript value needs to be modified before it is sent to the consumer.

[2] Not to be confused with the indirect access approach which is one approach to handle data structure in data-flow model of computation by providing special structure store facilities.

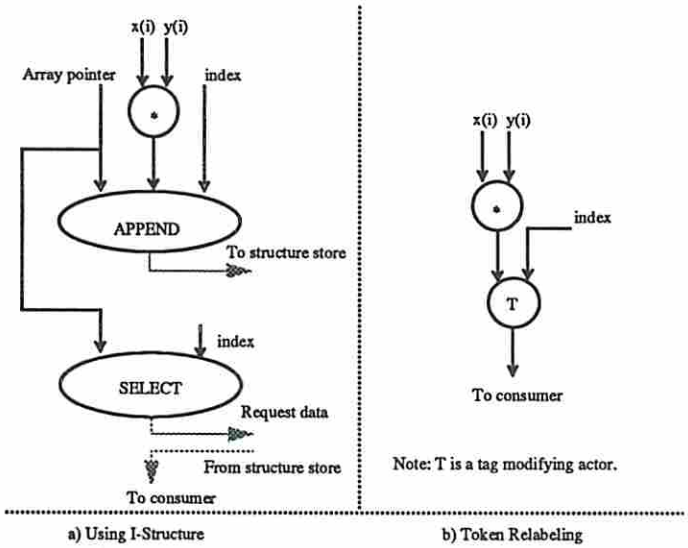

Figure 6.1: Data-flow graphs for $a(i) = x(i) \times y(i)$ when different schemes are used.

usually result in poor performance[3] when compared to that of dense matrices in all commercially available vector/parallel computers. There are three main reasons for poor performance in sparse computations as reported in [81]:

- Extra memory accesses due to array subscripts which are themselves arrays.
- Loss of data locality.
- Difficulty to detect parallelism due to array subscripts whose values may not be known at compile time.

Due to its robustness against memory latency, the data-flow model of computation is a natural choice for applications with irregular access patterns such as sparse computations. On the other hand, the von Neumann model's sensitivity to memory latency is further aggravated by the extra memory accesses required by indirect addressing [7, 81]. Furthermore, taking advantage of non-strict operations allowed by the Token Relabeling scheme on arrays makes the data-flow model of computation a good candidate particularly for numerical applications with irregular access patterns.

[3]Between 5 and 20 percent according to [81].

This paper concentrates on the issues of detection and exploitation of parallelism at run-time since in many instances it is impossible to detect parallelism at compile time when indirect addressing is used. There has been research on this issue by other researchers mainly for von Neumann type of multiprocessors [92, 98]. Discussion of these research is omitted in this paper due to limited space. This paper proposes a scheme which detects and exploits parallelism during run-time where structures are handled by Token Relabeling. The target architecture is assumed to be a data-flow multiprocessor which adheres to the MIT's dynamic data-flow model of execution [6].

In section two, the Token Relabeling method is explained. In section three, the basic idea of the proposed scheme is described with a simple example. In section four, the proposed scheme is described with some applications in which this scheme may be used to advantage. In section five, some preliminary performance result is discussed. Section six presents some concluding remarks along with comments on future research directions.

6.2 The Token Relabeling Method

As briefly mentioned in the previous section, Token Relabeling handles arrays by treating each array element like a scalar and therefore no special structure store facility is used. At the level of the data-flow graph, the notion of array is totally done away with. Each array element is carried in a scalar token which is identified by its tag[4]. A one dimensional array $A(i)$ of size $N, i = 0 \ldots N-1$ is represented by a set of scalar tokens, $\{A(i)_{[i]} \mid i = 0 \ldots N-1\}$, where the subscript represents the tag. The consumer of the array "picks" an array element from the set of tokens by providing a matching token of the desired index value. While Token Relabeling does not use special actors for array operations such as APPEND and SELECT, it requires a number of tag manipulating actors. Typical tag manipulating actors are shown in Figure 6.2. The X_{inc} actor functions exactly

[4]Actually, each array element does not have to be in a separate scalar token. A number of array elements may be carried in a single token if the underlying hardware supports vector operations.

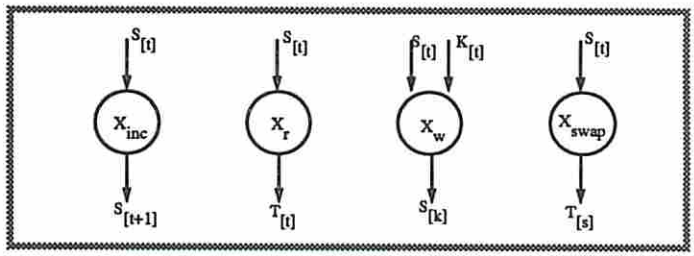

Figure 6.2: Some actors used in Token Relabeling scheme.

like a D actor of the U-Interpreter [6]. The functions of the other tag manipulating actors are as follows:

- X_r reads or copies the tag value into the data field. This can be used to find out the position of the array element in the array.
- X_w writes the data value into the tag field which implies changing the index value. This actor is usually used in scatter or gather operations.
- X_{swap} swaps the data and the tag values.

The Token Relabeling scheme provides non-strict operations on array elements as in the I-Structure method. This is another source of parallelism since array operations are allowed before all array elements are available. For example, $S(i)$ and $G(i)$ arrays representing subscripts in scatter and gather operations in Figure 6.3. The arrays can be forwarded to their respective consumers as each array element is calculated. By allowing non-strict operations, the latency caused by indirect addressing can be greatly reduced.

Deferred read requests of array elements provided by I-Structures can also be supported by the Token Relabeling scheme. A single deferred read request is supported by default because of the data-flow model of execution. The partner token of an array element which may not be available will reside in the token match unit until it is available. This token acts as the read request token used in the I-Structure. Multiple deferred read requests may be supported by extending the function of the token match unit at some cost.

An example of a scatter and a gather operation using this technique is shown in Figure 6.3. For the scatter operation $A(S(i)) = \alpha \times B(i)$, B elements are

scattered by simply being re-tagged by $S(i)$ in the X_w actor. The gather operation $A(i) = X(i) \times B(G(i))$, is a bit complicated by the process of re-tagging $B(G(i))$. $B(G(i))$ is originally tagged by $G(i)$, but since it must be matched with $X(i)$ for the multiplication operation it needs to be re-tagged by i. The re-tagging process involves swapping the data and the tag of the $G(i)$ tokens which are originally tagged by i resulting in a set of tokens, $\{i_{[G(i)]} \mid i = 1 \ldots N\}$. This swapping is done by X_{swap} actor. The set of tokens $\{i_{[G(i)]}\}$ are matched with $\{B(G(i))_{[G(i)]}\}$ for processing by the X_w actor to yield re-tagged tokens, $\{B(G(i))_{[i]}\}$.

Another argument for the Token Relabeling in the context of the topic of this paper is its flexibility in accommodating the extraction of *partial* parallelism during run-time. Partial parallelism is defined to be those cases where the loop bodies of some iterations may be executed in parallel while the other iterations must be executed in sequence due to data dependency.

On the other hand, it is unwieldy to use I-Structure especially if the data dependent portion of the loop updates an element of the array more than once. This is because the I-Structure does not allow writing to the same element more than once [9]. In such a case, there needs to be a logical partitioning of the I-Structure to accommodate the entire array. One I-Structure is used for the parallel portion and a new instance of the I-Structures is needed for each of those iterations that needs to be executed in sequence. While creating a new instance of the I-Structure for each of the sequential iteration, data from the previous iteration must be copied into the newly created I-Structure.

6.3 Basic Scheme

This section begins by introducing some examples where partial parallelism may exists. A simple run-time technique is introduced which can be used to exploit partial parallelism in the cases where an array element is updated only once during the entire iterations.

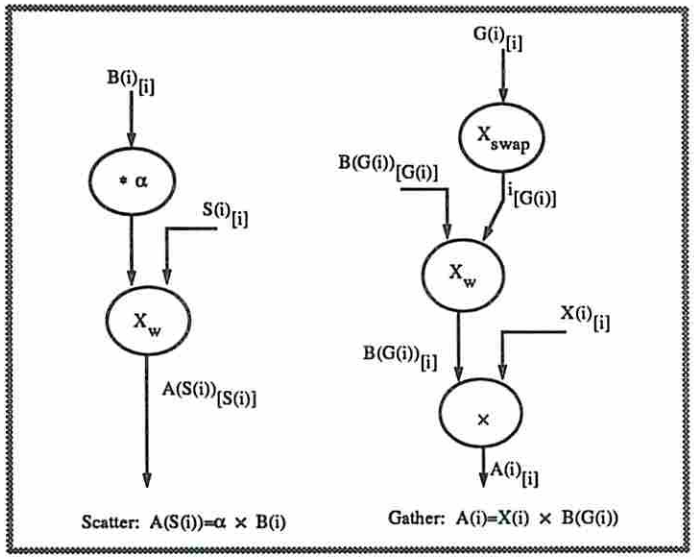

Figure 6.3: Scatter and gather operations using Token Relabeling method

i	0	1	2	3	4	5	6
digit[i]	3	3	4	2	3	1	5

Table 6.1: Sample array subscript values.

6.3.1 Basic Idea

An example of a situation where potential dynamic parallelism exists is the histogramming program shown in Figure 6.4. The program is written in SISAL (Stream and Iteration in a Single Assignment Language) [71]. The program loop increments the array elements of array `slot` at each iteration. This loop had to be written in a sequential loop construct because of possible recurring of array subscript values over the iteration. However, depending on the value of `digit[i]` for $0 \leq i \leq N-1$ where N is the size of the array, some of the iterations may be executed concurrently. Some sample values of `digit[i]` is shown in Table 6.1 for a case when $N = 7$.

From Table 6.1, it can be determined that the part of the program that needs to be executed sequentially are the iterations $0, 1$, and 4 due to the fact that array subscript `digit` recurs at those iterations. All other iterations may be executed in parallel since they are independent of each other. The values of `digit[i]`, however, are not known to the compiler and thus the loop will have to be executed

```
function Histo (N:integer; digit:OneDim returns OneDim)

  for initial
    slot := array_fill (1,N,0);
    i := 1
  while i <= N repeat
    slot := old slot [digit [old i]:
                      old slot [digit [old i]] + 1];
    i := old i + 1
  returns value of slot
  end for

end function
```

Figure 6.4: Histogramming program written in SISAL.

in a sequential manner if no other provisions are made to detect parallelism at run-time when the values of `digit[i]` become available.

As mentioned in the beginning of the paper, the structure handling scheme is assumed to be the Token Relabeling method. In this method, array elements are carried individually in a scalar token. Thus an array element that is produced is sent directly to its consumer instead of waiting for the whole array to be produced. The idea of run-time parallelism detection is to use this non-strictness to detect parallelism without incurring too much overhead.

In such cases as the histogramming example, the value of the array subscript `digit`, which itself is an array is needed to detect parallelism. It is possible to devise a mechanism which detects parallelism using the values of the array subscripts as they become available. This parallelism detecting mechanism executes in *parallel* with the loop in consideration. The mechanism controls the execution of the loop as dependencies are checked across iterations. Figure 6.6 shows the block diagram of the basic scheme which breaks a sequential loop which may contain a parallel portion and a data dependent portion which may contain some parallelism. The latter portion is controlled by the parallelism extraction mechanism. There are two scenarios in which no performance improvement can be obtained even though parallelism may exist:

```
function wavefront(N:integer;a:TwoDim returns TwoDim)

  for initial
    x_array := a;
    i := 1;
  while i < N
  repeat
    x_array :=
    for initial
      j := 1;
      w_array := old x_array;
    while j < N
    repeat
      w_array := old w_array[[old i,old j]: (old w_array[old i,old j-1] +
        old w_array[old i-1,old j] + old w_array[old i,old j+1] +
        old w_array[old i+1,old j])/4.0]
      j := old j + 1;
    returns value of w_array
    end for
    i := old i + 1;
  returns value of x_array
  end for
end function
```

Figure 6.5: Wavefront example written in SISAL.

1. The rate of array subscripts coming into the mechanism is slow compared to the sum of the overhead caused by the parallelism detection mechanism and the time for the array operations, *i.e.*, Average $T_{interval} > T_{overhead} + T_{arrayop}$.

2. The overhead caused by the parallelism detection mechanism is greater than the array modifying operation, *i.e.* $T_{overhead} > T_{arrayop}$.

6.3.2 Wavefront Example

In this section, a demonstration of the run-time parallelism exploitation is given on a relaxation example which possesses wavefront parallelism. Figure 6.5 is a SISAL representation of a 4-point relaxation problem.

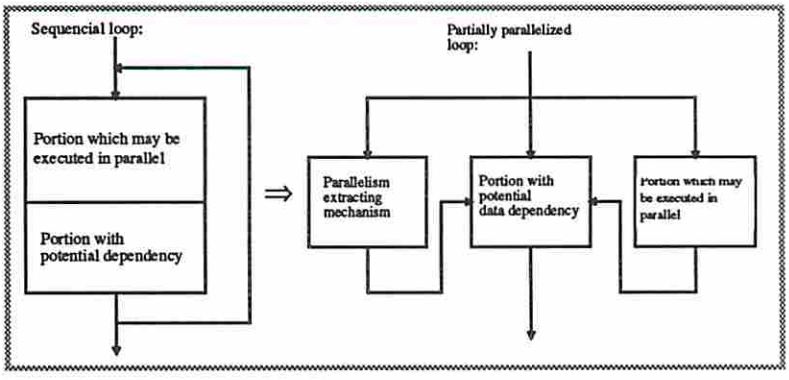

Figure 6.6: Block Diagram of the Basic Scheme.

The loop shown must be written in the sequential loop construct because of the data dependency between iterations in the operations on the array w_array. At each iteration (i, j), relaxation is done by using two updated array elements, i.e. w_array(i, j-1) and w_array(i-1, j) and two original array elements, i.e. w_array(i, j+1) and w_array(i+1, j). Thus it can be easily seen that the array elements on a 45 degree line can be calculated in parallel. Figure 6.8 shows the data-flow graph representation of the program along with the mechanism which allows such parallelism exploitation at run-time. The wavefront example was chosen because it is relatively a simple example. It does not mean, however, that the run-time exploitation of parallelism for such a case is a good one. Actually, compile time analysis would produce a better result for the wavefront example.

The parallelism detection and exploitation mechanism in the case of the wavefront example is very simple because the array indices do not repeat.This means that an array element is updated only once. The mechanism consists of a number of "multiplexer" actors and comparator actors. The original array elements are input to the **true** port of the multiplexer and the updated array elements are input to the **false** port of the multiplexer. Predicate tokens for the multiplexers are provided by the comparator actors which compare the indices of the array elements which are to be consumed and the indices of the array elements which are to be updated. Thus the array index (i, j) is compared with indices (i, j-1), (i-1, j), (i, j+1) and (i+1, j). If (i, j) is greater than the index being compared, a **false** token is generated which means that the updated value of the array element should be used in the computation. The updated array elements are fed back to

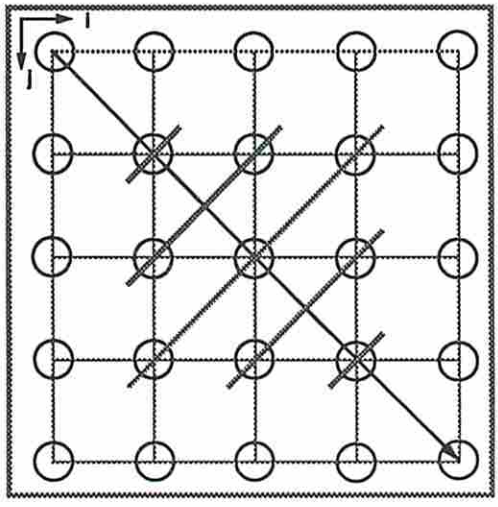

Figure 6.7: Wavefront parallelism

the **false** inputs of the multiplexers. The consumption and the production of array elements are fully synchronized by the graph to exploit all possible parallelism while respecting data dependencies.

The extra mechanism that needs to be inserted in the original data-flow graph is not always simple as shown in the previous example. If the array has a recurrence and the order of computations of the array elements which recur are important, the mechanism becomes more complex. In the next section, a scheme to exploit parallelism during run-time is presented which handles this general case.

6.4 General Scheme

Based on the idea introduced in the previous section, this section proposes a general scheme. This scheme can be applied to any loop which may possess some partial parallelism which can only be determined at run-time.

6.4.1 Justification

One application area where dynamic parallelism exploitation is needed is in computational fluid dynamics. Solving steady state solutions of the Euler equations

about objects of complex geometry in many important computational fluid dynamics applications has been hampered by the ability to generate adequate meshes about such objects. One approach that is gaining popularity is to use completely unstructured meshes where each mesh is a triangular element in two dimensions [69].

While using such meshes results in better accuracy and flexibility, the computation time suffers due to the indirect addressing of array elements. When structured rectangular grids are used it is a simple matter to determine adjacency relations between volume elements by simple incrementing or decrementing of indices [40]. However, the numbering of nodes and cells in unstructured grids are random. In addition, since the fluxes are computed across each edges of the mesh, the edges become the indices of the array where each array elements are 4-tuples of $(N1, N2, I1, I2)$, where $N1$ and $N2$ are nodes on each end of the edge represented by the index. $I1$ and $I2$ refer to the volume elements on either side of the edge [69, 13].

Indirect addressing is used in a loop when volume elements are accessed to compute flux balance over the entire flow field. Since each volume element receives contributions from more than one edge, the loop will have a recurrence. Analysis of such a loop at compile time does not give any information about parallelism since the values of 4-tuples will only become available at run-time. A typical method currently used by most applications running on vector computers is to sort the indices such that each vector group does not have a recurrence before the loop is entered.

6.4.2 Proposed Scheme

The proposed scheme is logically divided into two parts which consist of a *dependency detector* and a *parallelism exploiter*. In the previous example of the wavefront method, comparator actors were the dependency detectors while the multiplexer actors were the parallelism exploiters. In the example, the comparators compared the data values of the matching tokens and generated the predicate tokens which specify whether the updated values or the initial values of array elements are to be consumed by the subsequent computations. The matching tokens

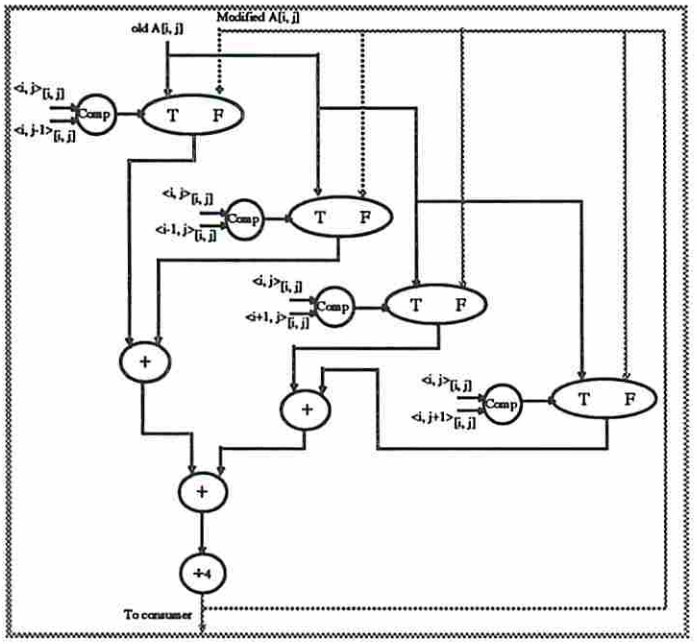

Figure 6.8: Data-flow graph which allows run-time parallelism exploitation.

were tagged by the i and j indices. This worked appropriately for the example because each array element was modified only once. However, when there is a possibility of recurrence in the loop meaning that an array element may be modified more than once, such a simple mechanism is not sufficient.

A loop which possesses a potential recurrence can be further classified as *order insensitive* or *order sensitive*. An example of an order insensitive loop is when the loop contains the following statement, $A(B(i)) = A(B(i)) + X(i)$. If $B(1) = B(7) = B(10) = 3$ and $A(3)$ was initialized to 0, then at the end of the iteration, $A(3) = X(1) + X(7) + X(10)$. Since the add operation is commutative, any of the 6 possible sequences will produce the same result. On the other hand, the following loop is order sensitive, $A(B(i)) = X(i) - A(B(i))$. Such a statement in a loop with recurrence will certainly produce different results depending on the sequence since subtraction is not a commutative operation.

If a loop is order sensitive, the process of the dependency detector must be sequential. The algorithm is as follows where $B(i)$ is the array subscript:

while $i < N$ do

$k = old\_updat(B(i))$;

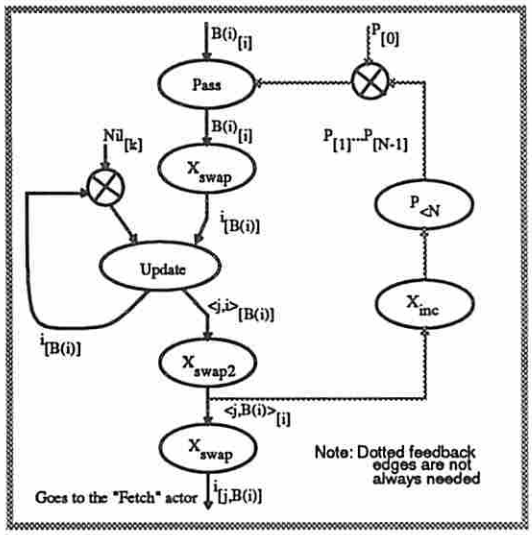

Figure 6.9: Data-flow graph representation of the dependency detector.

if $old\_updat(B(i)) = nil$ then
  consume the initialized array element $A(B(i))$
else
  consume $A(B(i))$ which was updated at iteration
  old_updat(B(i))
$old\_updat(B(i)) = i$;
end while;

Note that the token which is produced by the dependency detector was tagged by the subscript value of the current iteration in the wavefront method example. In the case of a loop with recurrence, however, this is not sufficient because the subscript values are not unique anymore. In the proposed scheme, each output token of the dependency detector has the form $i_{[j,B(i)]}$ which makes each token unique. The tag field consists of two integer fields where the first field corresponds to the iteration where the array element was last updated and the second field identifies the array element which the current iteration is required to consume. The data value is also an integer which identifies the current iteration. The parallelism exploiter picks the matching array element and re-tags the first field

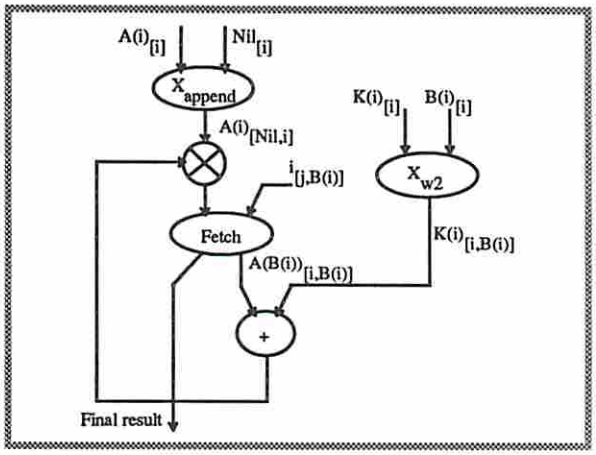

Figure 6.10: Data-flow graph representation of the parallelism exploiter.

of the tag with the value that is in the data field before sending it to the consumer of the array element.

Figure 6.9 shows the data-flow graph of the dependency detector. The left input of the Update actor is initialized to *nil*. After a pair of tokens are matched for the Update actor, the left input token is fed back such that left input token $= i_{[B(i)]}$ when the matching tokens were such that the left input token $= j_{[B(i)]}$ and the right input token $= i_{[B(i)]}$. Thus, the left input port of the Update actor holds the history of which particular array element was most recently modified. The Update actor forwards the two data values of the input tokens, $< i, j >_{[B(i)]}$ to the destination actor. X_{swap2} actor is similar to the X_{swap} actor except that the second field of the data field gets swapped with the tag. The outer feedback loop, (dotted line) is not needed when the loop is order insensitive.

Figure 6.10 shows the data-flow graph of the parallelism exploiter. The X_{append} actor re-tags the array elements into the form to be matched by the tokens sent by the dependency detector. Tokens sent by the dependency detector are destined to the Fetch actor which forwards the matching array element to the subsequent computation after re-tagging. The net result of using the mechanisms described above is the transformation of a purely sequential loop into a partially sequential loop.

An example using the same sequence as in Table 6.1 of the histogramming example is given, *i.e.* array B of the dependency detector corresponds to array

digit of the histogramming program. Assume that the order of array subscript $B(i)$ arriving at the input of dependency detector is as follows:

$B(3)_{[3]} \rightarrow B(0)_{[0]} \rightarrow B(4)_{[4]} \rightarrow B(1)_{[1]} \rightarrow B(6)_{[6]} \rightarrow B(2)_{[2]} \rightarrow B(5)_{[5]}$

If the loop is order sensitive, the order of the tokens generated by the dependency detector will be as follows:

$0_{[nil,B(0)]} \rightarrow 1_{[0,B(1)]} \rightarrow 2_{[nil,B(2)]} \rightarrow 3_{[nil,B(3)]} \rightarrow 4_{[1,B(4)]} \rightarrow 5_{[nil,B(5)]} \rightarrow 6_{[nil,B(6)]}$

The tokens which correspond to iterations which are determined to be independent, *i.e.* can be executed in parallel have the first field of the tag as *nil*. In this example, iterations $i = 2, 3, 5, 6$ are independent of each other and iterations, $i = 0, 1, 4$ are dependent of each other. The first tag field of each token corresponding to this dependent case indicates the iteration which it depends on. The token which corresponds to iteration 1 has the value 0 in the first field of its tag. This is correct since iteration 1 can only proceed after iteration 0 has been completed. The same is true for iteration 4 which is dependent on iteration 1.

If the loop is order insensitive, the order of the tokens generated by the dependency detector is as follows assuming the arriving order of array subscripts to be as given above.

$3_{[nil,B(3)]} \rightarrow 0_{[nil,B(0)]} \rightarrow 4_{[0,B(4)]} \rightarrow 1_{[4,B(1)]} \rightarrow 6_{[nil,B(6)]} \rightarrow 2_{[nil,B(2)]} \rightarrow 5_{[nil,B(5)]}$

For order insensitive loops, tokens arriving at the dependency detector are not delayed as is the case for order sensitive loops. Tokens are generated in the order input tokens arrive. This will decrease the overhead created by the dependency detector. The tokens generated by the dependency detector are sent to the parallelism exploiter which forwards the matching array element tokens to those subsequent operations which consume the array elements. Operations in independent iterations can immediately be activated without waiting for the result of the previous iteration.

6.4.3 Enhancement

It can be a problem if the rate of arrival of the array subscript to the dependency detector is very fast compared to the throughput of the dependency detector.

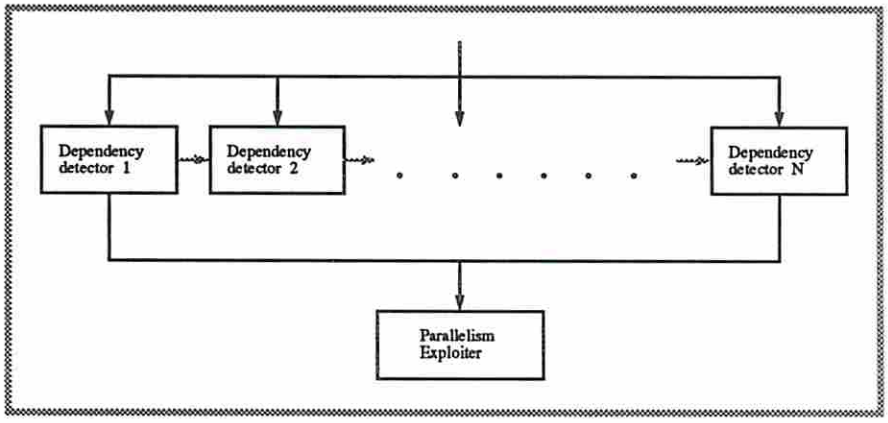

Figure 6.11: Partitioning of dependency detector.

In such a case the sequential processing of the dependency detector becomes a bottleneck such that it is possible that no parallelism will be exploited after a while if the number of loop iterations is very large. Such a situation may arise for example, in the computational fluid dynamics if a parallel grid generation algorithm is used.

To overcome this problem, partitioning of the dependency detector is suggested. Each dependency detector handles a subset of array subscripts such that each detector processes N/k array subscripts. N is the total number of array subscripts and k is the number of partitions. Each partition of detectors checks for data dependency within the given array subscripts. If the loop is order insensitive, performance proportional to the number of partition can be expected. If the loop is order sensitive, each partition must wait for the previous partition to provide dependency information before outputting to the exploiter.

A performance gain can still be achieved even in these cases because dependency detection can still be performed within the partition and final decision can be made as soon as the needed information is provided by the previous partition. This is analogous to the idea of carry lookahead adder where each carry lookahead circuit computes carry propagate and carry generate terms and the carry terms are computed as soon as carry input is available from the adjacent lookahead circuit.

6.5 Preliminary Simulation Results

Preliminary simulations have been performed using a micro data-flow simulator. The simulator provides a maximum of 64 processors interconnected by a hypercube. Each functional block of a processing unit in the simulated machine is assumed to process a token in 1 time unit. Furthermore, each hop between two nodes in the network by a token also adds 1 time unit.

The simulation is performed using a small array size of 128 and the performance measurement is done using 2 different cases. The first case is where each array element is updated once and consumed once. The second case is where each array element is updated once, but consumed multiple times. The wavefront example belongs to the second case since an array element is modified only once, but consumed multiple times when other array elements are evaluated.The values of array subscripts have been generated by a random number generator provided by the Unix operating system.

Figures 6.12 and 6.13 show the simulated execution times and the speedup achieved for the 2 cases using different number of processors along with the execution time when the loop is executed in a sequential manner.[5] The first case yields the maximum performance, however, this should not be viewed as a general performance expected of run-time extraction of parallelism. Performance result of the first case only shows the upper bound performance improvement. The simulated execution time for the second case is longer as expected. This is due to the fact that the second case requires a more complex mechanism compared to the first case. In both cases, saturation is reached around the region where the number of processors are between 8 and 10. This is due to the small size of the array used in the simulation. In other words, due to small array size, there isn't enough parallelism to be fully utilized by the available processors as the number of available processors are increased. The saturation point will move towards the right of the x-axis as larger arrays are used.

[5]The sequential execution time increases steadily as more processors are used. This is due to the communication time overhead caused by the array elements being moved across different processors.

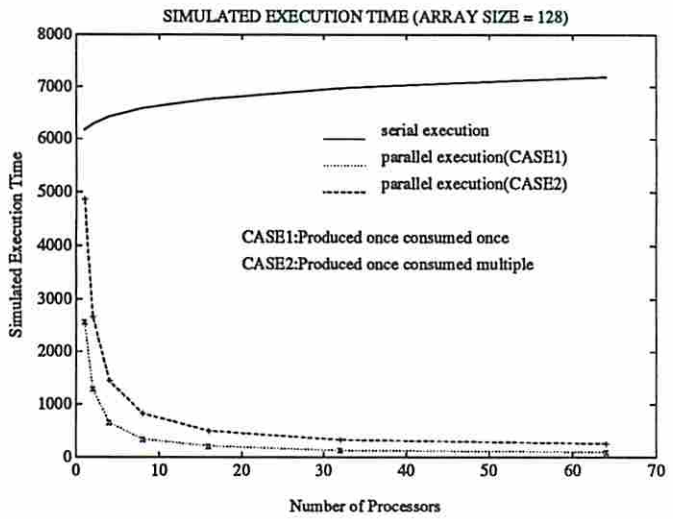

Figure 6.12: The simulated execution time when the array size is 128.

6.6 Conclusion and Future Research Issues

There are many real world numerical applications where the existence of parallelism can only be determined during run-time when necessary information becomes available. A scheme which allows exploitation of parallelism of a sequential loop during run-time has been presented. The model of computation is assumed to be data driven and the structure handling scheme is Token Relabeling where each array element can be treated as a single entity. The proposed mechanism which allows exploitation of parallelism is executed in parallel to the loop under consideration and controls the execution of the loop.

Sequential loops can be classified into two types, *i.e.*, order sensitive and order insensitive. Such a classification is beneficial in data-flow computations because the arrival order of data destined for an actor is not known and taking advantage of such occurrences can improve performance. If a loop is order sensitive, iterations where a data dependency exist must be executed in order. A dependency detector of this type tends to be less efficient because of the necessary feed back loop in the graph. Parallelism exploitation is more likely for the order insensitive loops because dependency detector is more efficient and thus contributes less overhead.

An enhancement idea has been proposed for cases when iteration length is large. The idea is to partition the dependency detector such that each detector

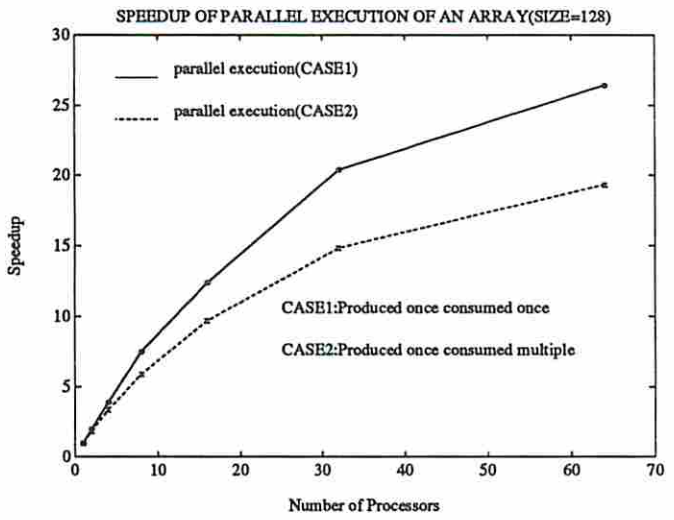

Figure 6.13: The speedup achieved when the array size is 128.

processes a subset of the loop. Order insensitive loops will work well with partitioning. Order sensitive loops will also gain because dependency detection within partition can still be done in parallel although final determination can only be made when the necessary information is sent from the previous partition.

Some small simulations have been performed for the general case where the correct working of the data-flow graph representations of the dependency detector and the parallelism exploiter have been observed. Large size simulation using some real application kernel is planned to better understand and determine the possible effect of the scheme on the overall performance of the application. Currently, SISAL is assumed to be the programming language of choice. The SISAL compiler outputs an intermediate form which is in the form of data-flow graph called, IF1. Work is in progress in translating the IF1 graph into the target machine which is a macro data-flow computer. Ideas presented in this paper are planned to be incorporated into this work.

Chapter 7

Summary and Conclusions

The objective of this thesis was to demonstrate the viability of functional programming in the context of parallel computing and propose a general framework for migrating SISAL programs to massively parallel distributed memory architectures. This chapter lists the research contributions and suggests future research issues.

7.1 Summary of Contributions

- Through an experiment, we successfully demonstrated that functional programming can indeed provide programmability as well as performance. We showed that it was possible to develop efficient parallel programs according to the following procedure:

    1. Write SISAL programs without being concerned with manual parallelization.
    2. Debug on a sequential machine and verify that the programs work correctly. The important point here is that potentially hazardous parallel debugging is not necessary.
    3. Once a program is verified to work correctly on a serial machine, compile the program without modification on parallel machines such as CRAY Y-MP, Sequent Balance, and Silicon Graphics.

We have observed that the Optimizing SISAL Compiler efficiently parallelized/vectorized programs through performance evaluations on a number of parallel machines.

On the other hand, programs written in imperative languages such as FORTRAN and C performed slightly better on sequential machines, but performed quite poorly compared to the SISAL programs.

- As part of demonstrating the viability of functional programming, a number of popular numerical methods are implemented to solve elliptic partial differential equations. They are multigrid method based on a full V-cycle, preconditioned conjugate gradient method, and symmetric domain decomposition method.

- To address the latency problem in distributed-memory parallel machines, we have chosen a fine-grain multithreaded execution model. However, conventional code block activation scheme has potentially high activation creation overhead and can cause frequent context switches. A new activation management technique called the Multiple Iterations per Activation (MIpA) is proposed which reduces activation creation overhead by allowing multiple activations to be created with a single resource manager service call. In addition, by allowing multiple instances of threads to coexist on a processor, there is less probability of context switches. The net effect is that, given a processor, more parallelism can be exploited with less overhead.

- Based on the MIpA technique, an efficient array handling scheme was proposed which can be used when the life-time of an array is close to those of its producer and consumer(s). The Direct Array Injection Technique (DAIT) directly forwards the produced array elements to consumer(s) without first storing the array in a global memory space. The advantage of this technique is that dynamic memory allocation/deallocation from global heap is avoided and the number of potential remote memory accesses is reduced.

- Based on the MIpA and the DAIT techniques, a new loop fusion optimization scheme was proposed which fuses the array producer and the consumer

loops into a single loop. The effect of this optimization is that loop body activation creation overhead and the possibility of context switches is greatly reduced.

- In many real scientific applications, it is difficult and in many cases impossible to determine at compile-time whether a loop can be executed in parallel. This is due to the fact that the value of an array subscript is not known at compile-time. At run-time, it may turn out that a significant amount of parallelism exists. Based on the Token Relabeling method, a run-time parallelism detection and exploitation technique is proposed in the context of dynamic data-flow execution model.

7.2 Future Research Issues

While working to address the research objectives as stated in this thesis, many interesting points have appeared which warrant further research. Their description is as follows:

Instruction level parallelism. The current implementation of the Optimizing SISAL Compiler does not exploit instruction level parallelism. The valuable information on the data dependency relationship among simple nodes is lost as the intermediate representation is translated into C. As more and more powerful microprocessors based on superscalar architectures appear, there is a strong incentive to exploit instruction level parallelism.

Software tools. To study various effects of compile/run-time techniques, extensive benchmarking is required. Currently, this is not possible due to lack of fully automatic path from the frontend to the code generator (for the simulator). In addition, to determine the appropriate size of the processor state and to understand the incremental context switch mechanism, a detailed simulator is needed which models the processor with the required architecture support for the MIpA model.

Applications development. All the partial differential equation solvers developed during the thesis work are based on the Finite Difference Method (FDM) of discretization. There are other approaches to solve partial differential equations based on the Finite Element Method (FEM) and Finite Volume Method (FVM) of discretization. Programs developed using FEM and FVM display different characteristics compared to those developed using FDM. Applications based on FEM and FVM need to be developed to examine the power and the limitations of the SISAL language as well as OSC. Knowledge gained should be used to extend the language and improve the compiler.

Distributed computing. Because no extra investment is required, distributed computing based on a network of workstations is becoming popular. Software packages such as PVM, PCN, and `p4` support this kind of parallel computing by providing a standard set of library routines. The disadvantage is that it is still programmer's responsibility to parallelize. It should be possible to compile SISAL programs for execution on a network of workstations by generating C code using the library functions provided by the packages.

Dynamic parallelism. The run-time parallelism detection and exploitation technique can benefit the performance of many real scientific applications. The proposed technique is based on a dynamic data-flow execution model. As the execution model has evolved to multithreaded model, the technique needs to be re-addressed to reflect the requirements of the multithreaded execution model.

Bibliography

[1] D. Abramson and G. Egan. Design of a high performance data-flow multiprocessor. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 4, pages 121–141. Prentice Hall, 1991.

[2] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6. ACM, ACM Press, June 1990.

[4] M. Amamiya. An ultra-multiprocessing architecture for functional languages. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 3, pages 95–119. Prentice Hall, 1991.

[5] Arvind, L. Bic, and T. Ungerer. Evolution of data-flow computers. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 1, pages 3–33. Prentice Hall, 1991.

[6] Arvind and K. Gostelow. The U-interpreter. *IEEE Computer*, 15(2):42–49, February 1982.

[7] Arvind and R. Iannucci. Two fundamental issues in multiprocessing. In *Conference on Parallel Processing in Science and Engineering*, 1987.

[8] Arvind and R. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. In *Parallel Architectures and Languages Europe*. Springer-Verlag, August 1987.

[9] Arvind and R. Thomas. I-structures: An efficient data type for functional languages. Technical Report LCS/TM-178, MIT, Laboratory for Computer Science, June 1980.

[10] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[11] M. Beckerle. An overview of the START(*T) computer system. Technical Report MCRC-TR-28, Motorola Inc., Cambridge Research Center, One Kendall Square, Building 200 Cambridge MA 02139, July 1992.

[12] G. Bell. Ultracomputers: A teraflop before its time. *Communications of the ACM*, 35(8):26–47, August 1992.

[13] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. Technical Report 90-41, ICASE NASA Langley Research Center, May 1990.

[14] G. Birkhoff and R. Lynch. *Numerical Solution of Elliptic Problems.* SIAM Press, 1984.

[15] A. Böhm, D. Cann, D. Grit, J. Feo, and R. Oldehoeft. *Draft SISAL Reference Manual Language Version 2.0.*

[16] A. Böhm and J. Sargeant. Code optimization for tagged-token dataflow machines. *IEEE Transactions on Computers*, 38(1):4–14, January 1989.

[17] W. Briggs. *A Multigrid Tutorial.* SIAM, 1987.

[18] R. Butler and E. Lusk. Monitors, Messages, and Clusters: the p4 Parallel Programming System. *To Appear in the Journal of Parallel Computing*, 1994.

[19] D. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, 1989.

[20] D. Cann. *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550, 1992.

[21] D. Cann. Retire Fortran: A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[22] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[23] CRAY Research Inc. CRAY *Standard C Programmer's Reference Manual*, SR-2074 1.1 edition, 1990.

[24] CRAY Research Inc. *CF77 Compiling System, Volume 1: Fortran Reference Manual*, SR.3071 5.0 edition, 1991.

[25] D. Culler, S. Goldstein, K. Schauser, and T. von Eicken. TAM-a compiler controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, July 1993.

[26] D. Culler and G. Papadopoulos. The Explicit Token Store. In J-L. Gaudiot and L. Bic, editors, *Journal of Parallel and Distributed Computing, Special Issue:Data-Flow Processing*, pages 289–308. Academic Press, December 1990.

[27] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *ASPLOS-IV Proceedings*, pages 164–175. ACM and IEEE, ACM Press, April 1991.

[28] D. Culler, K. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In M. Cosnard, K. Ebcioğlu, and J-L. Gaudiot, editors, *IFIP Transactions: Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 153–164. IFIP, North-Holland, January 1993.

[29] W. Dally, J. Fiske, J. Keen, R. Lethin, M. Noakes, P. Nuth, R. Davison, and G. Fyler. The Message-Driven Processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–38, April 1992.

[30] J. Dennis. Dataflow supercomputers. *Computer*, 13(11):48–56, November 1980.

[31] J. Dennis. The evolution of static data-flow architecture. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2, pages 35–91. Prentice Hall, 1991.

[32] K. Diefendorff and M. Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[33] J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7(2):166–175, January 1993.

[34] K. Ekanadham. A perspective on Id. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 6, pages 197–253. ACM Press, 1991.

[35] D. Engelhardt and A. Wendelborn. Investigating the memory performance of the optimising SISAL compiler. In J. Feo, C. Frerking, and P. Miller, editors, *Proceedings of the Second Sisal Users' Conference*, pages 257–270. Lawrence Livermore National Laboratory, December 1992.

[36] P. Evripidou and J-L. Gaudiot. A decoupled graph/ computation data-driven architecture with variable-resolution actors. In B. Wah, editor, *Proceedings of 1990 International Conference on Parallel Processing : Vol 1 Architecture*, pages 405–414. The Pennsylvania State University, The Pennsylvania State University Press, August 1990.

[37] J. Feo. The Livermore Loops in SISAL. Technical Report UCID-21159, Lawrence Livermore Mational Laboratory, August 1987.

[38] J. Feo. SISAL. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing*, chapter 9, pages 337–386. North-Holland, 1992.

[39] J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, December 1990.

[40] C. Fletcher. *Computational Techniques for Fluid Dynamics Volume 1*, chapter 5. Springer-Verlag, 1988.

[41] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *To Appear in Scientific Programming*, 1994.

[42] G. Gao, H. Hum, and J-M. Monti. Towards an efficient hybrid dataflow architecture model. In E. Arts, J. van Leeuwen, and M. Rem, editors, *PARLE'91 Parallel Architectures and Languages Europe*, pages 355–371. Springer-Verlag, June 1991.

[43] J-L. Gaudiot. Structure handling in data-flow systems. *IEEE Transactions on Computers*, C-35(6):489–502, June 1986.

[44] J-L. Gaudiot and M. Ercegovac. Performance evaluation of a simulated data-flow computer with low-resolution actors. *Journal of Parallel and Distributed Computing*, 2:321–351, 1985.

[45] J-L. Gaudiot and C. Kim. Data-driven and multithreaded architectures for high-performance computing. In T. Casavant and P. Tvrdik, editors, *Parallel Computers: Theory and Practice*, chapter 4. IEEE Computer Society Press, In press.

[46] J-L. Gaudiot and Y. Wei. Token relabeling in a tagged-token data-flow architecture. *IEEE Transactions on Computers*, 38(9), 1989.

[47] G. Golub and C. van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[48] V. Grafe and J. Hoch. The Epsilon-2 multiprocessor system. *Journal of Parallel and Distributed Computing*, 10:309–318, December 1990.

[49] V. Grafe, J. Hoch, G. Davidson, V. Holmes, D. Davenport, and K. Steele. The Epsilon project. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 6, pages 175–205. Prentice Hall, 1991.

[50] J. Gurd, C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[51] J. Gustafson. Reevaluating amdahl's law. *Communication of ACM*, 31(5):532–533, May 1988.

[52] M. Haines and W. Böhm. A virtual shared addressing system for distributed memory SISAL. In *Proceedings of the Third Sisal Users' Conference*. Lawrence Livermore National Laboratory, October 1993.

[53] J. Hicks, D. Chiou, B. Ang, and Arvind. Performance studies of the Monsoon dataflow processor. *Journal of Parallel and Distributed Computing*, July 1993.

[54] W. Hillis and L. Tucker. The CM-5 connection machine: A scalable supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[55] K. Hiraki, S. Sekiguchi, and T. Shimada. Status report of SIGMA-1: A dataflow supercomputer. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 7, pages 207–223. Prentice Hall, 1991.

[56] K. Hiraki, T. Shimada, and K. Nishida. A hardware design of the SIGMA 1- a data flow computer for scientific computations. In *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984.

[57] T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata. Improving AP1000 parallel computer performance with message communication. In *The 20th Annual International Symposium on Computer Architecture*, volume 21, pages 314–325. ACM and IEEE, ACM Press, May 1993.

[58] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[59] R. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988.

[60] R. Iannucci. *Parallel Machines: Parallel Machine Languages, The Emergence of Hybrid Dataflow Computer Architecture*. Engineering and Computer Science. Kluwer Academic Publishers, 1990.

[61] M. Kallstrom and S. Thakkar. Programming three parallel computers. *IEEE Software*, pages 11–22, January 1988.

[62] C. Kim and J-L. Gaudiot. A scheme to extract run-time parallelism from sequential loops. In *Proceedings of the 5th ACM International Conference on Supercomputing*. ACM, ACM Press, June 1991.

[63] C. Kim and J-L. Gaudiot. Parallel computing with the SISAL functional language: Programmability and performance issues. *Submitted to Software - Practice and Experience*, 1993. Revised and resubmitted.

[64] C. Kim and J-L. Gaudiot. Programmability and performance issues: the case of an iterative partial differential equation solver. In *Proceedings of the Third Sisal Users' Conference*. Lawrence Livermore National Laboratory, October 1993.

[65] C. Kim and J-L. Gaudiot. A direct array injection technique in a fine-grain multithreading execution model. In *Proceedings of the 8th International Parallel Processing Symposium*. IEEE Computer Society, IEEE Computer Society Press, April 1994. To appear.

[66] C. Kim and J-L. Gaudiot. A hierarchical activation management technique for fine-grain multithreading execution. In *PARLE'94 Parallel Architectures and Languages Europe*. Springer-Verlag, July 1994. To appear.

[67] S. Komori, K. Shima, S. Miyata, T. Okamoto, and H. Terada. The data-driven microprocessor. *IEEE Micro*, 9(3):45–59, June 1989.

[68] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[69] D. Mavriplis. Multigrid solution of the two-dimensional Euler equations on unstructured triangular meshes. *AIAA Journal*, 26, 1988.

[70] J. McGraw, D. Kuck, and M. Wolfe. A debate: Retire FORTRAN? *Physics Today*, 37(5):66–75, May 1984.

[71] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL Language Reference Manual Version 1.2*, March 1985.

[72] S. Mitrovic. An IF2 code generator for ADAM architecture. In J. Feo, C. Frerking, and P. Miller, editors, *Proceedings of the Second Sisal Users' Conference*, pages 93–109. Lawrence Livermore National Laboratory, December 1992.

[73] R. Nikhil. Id (version 90.1) reference manual. Technical Report CSG Memo 284-2, MIT, July 1991.

[74] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[75] R. Nikhil and Arvind. Id: a language with implicit parallelism. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing*, chapter 5, pages 169–215. North-Holland, 1992.

[76] R. Nikhil, G. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 1992. ACM, ACM Press.

[77] H. Nishikawa, H. Terada, S. Komori, K. Shima, T. Okamoto, and S. Miyata. Architecture of a VLSI-oriented data-driven processor: the Q-v1. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 9, pages 247–264. Prentice Hall, 1991.

[78] R. Oldehoeft and D. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, 1988.

[79] J. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Frontiers of Computer Science. Plenum Press, 1988.

[80] C. Rao and L. Kang. Symmetric domain decomposition and symmetric space decomposition. In *Proceedings of the International Conference on Scientific Computation.* World Scientific Press, 1992.

[81] Y. Saad and H. Wijshoff. A benchmark package for sparse matrix computations. In *1988 International Conference on Supercomputing.* ACM, ACM Press, 1988.

[82] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. Design of the dataflow single-chip processor EMC-R. *Journal of Information Processing*, 13(2):165–173, 1990.

[83] J. Sargeant and C. Kirkham. Stored data structures on the Manchester dataflow machine. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 235–242. ACM and IEEE, ACM Press, June 1986.

[84] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 146–155. ACM, ACM Press, May 1992.

[85] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi. Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 226–234. ACM and IEEE, ACM Press, June 1986.

[86] T. Shimizu, T. Horie, and H. Ishihata. Low-latency message communication support for the AP1000. In *The 19th Annual International Symposium on Computer Architecture*, volume 20, pages 288–297. ACM and IEEE, ACM Press, May 1992.

[87] S. Skedzielewski. SISAL. In B. Szymanski, editor, *Parallel Functional Languages and Compilers*, chapter 4, pages 105–157. ACM Press, 1991.

[88] S. Skedzielewski and J. Glauert. *IF1 An Intermediate Form for Applicative Languages.* Computing Research Group, Lawrence Livermore National Laboratory, P.O. Box 808, L-306, Livermore, CA 94550, 1985.

[89] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods.* Oxford Applied Mathematics and Computing Science Series. Oxford University Press, 1985.

[90] V. Srini. An architectural comparison of dataflow systems. *Computer*, pages 68–88, March 1986.

[91] T. Sterling and J. Arnold. Fine grain dataflow computation without tokens for balanced execution. *Journal of Parallel and Distributed Computing*, 18(3):327–339, July 1993.

[92] P. Tang, P. Yew, and C. Zhu. Compiler techniques for data synchronization in nested parallel loops. In *1990 International Conference on Supercomputing.* ACM, ACM Press, 1990.

[93] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM and IEEE, ACM Press, May 1992.

[94] I. Watson and J. Gurd. A practical data-flow computer. *IEEE Computer*, 15(2):51–57, February 1982.

[95] Y. Wei and J-L. Gaudiot. Compiling programs to direct access data-flow graphs. In *Proceedings of 1990 International Conference on Parallel Processing*, 1990.

[96] M. Welcome, S. Skedzielewski, R. Yates, and J. Ranelletti. *IF2: An Applicative Language Intermediate Form with Explicit Memory Management.* University of California Lawrence Livermore National Laboratory, December 1986.

[97] A. Wendelborn and H. Garsden. Exploring the stream data type in SISAL and other languages. In M. Cosnard, K. Ebcioğlu, and J-L. Gaudiot, editors, *IFIP Transactions: Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 283–294. IFIP, North-Holland, January 1993.

[98] C. Zhu and P. Yew. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Transactions on Software Engineering*, SE-13, 1987.