A Framework for Coarse Grain Parallel Execution of Functional Program

Dae-Kyun Yoon

CENG Technical Report 96-08

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

April 1996

A FRAMEWORK FOR COARSE GRAIN PARALLEL EXECUTION OF FUNCTIONAL PROGRAMS

by

Dae-Kyun Yoon

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)

April 1996

Dedication

To my wife Kyoung-Ran and my sons Doheny and Justin. Without your love and understanding, I would never have been able to finish the program.

Acknowledgements

I am indebted to my advisor, professor Jean-Luc Gaudiot, for his inspiration, guidance, and support. It was a privilege for having been one of his students. I would also like to express my gratitude to professors Sandeep Gupta and Ellis Horowitz for serving on my dissertation committee. Their inquisitive questions have stimulated my research. I thank professor Viktor Prasanna and Dr. Paul Suhler for being on my Ph.D guidance committee.

I would like to thank my former group members Professors Chinhyun Kim Andrew Sohn, and Dr. Chih-Ming Lin for giving me advice and encouragement. I thank my colleagues Namhoon Yoo, Hiecheol Kim, Moez Ayed and James Burns who have become my good friends over the years. Numerous discussions I had with them have stimulated my research greatly. I also acknowledge group members Yung-Syau Chen, Steve Jenks, Wen-yen Lin, Chulho Shin, Chung-Ta Cheng and Hung-Yu Tseng. Special thanks goes to Mary Zittercob, Rohini Montenegro, and Joanna Wingert for their assistance.

Many thanks are due to my friends Donghyun Heo, Kangwoo Lee, Joonho Ha, Jaeheon Jung, Edward Kim, Sanghoe Koo and Seungho Cha for their friendship and sparing their valuable time in many occasions. I would also like to express special thank to professor Yong-Doo Lee at Taegu university and his family for their affection and support for our family.

My special gratitude goes to my mother-in-law who has supported my family in many ways during my study at USC. At last, but not least, I thank my parents for raising and educating me. Without them, I would not be here.

Contents

List Of Tables

List Of Figures

Abstract

During the past decade, many commercial multiprocessor systems and software tools have been introduced for parallel application development. However, traditional sequential programming in imperative languages renders many problems with parallel programming in terms of the *programmability* due to the complexity incurred by the parallelism control. On the contrary, in a pure functional programming language, parallelism needs not be expressed by the programmer since it is implicit. Each statement can be executed in any order as long as data dependencies are respected. However, a functional language alone will not guarantee ideal parallel performance. There are still many issues including algorithm design, high-level programming and compilation that remain to be investigated.

This thesis addresses the issues of effective parallel execution of functional programs on various types of target architectures. Our approach is to exploit *coarse grain* parallelism. The coarse-grain approach is better suited in conventional multiprocessor systems when the synchronization cost is a significant factor in the overall cost of executing a program. The appropriate abstract execution model and the optimization schemes for the coarse-grain parallel execution have been thus developed. We have extended the runtime system of OSC (Optimizing Sisal Compiler) in order to implement the proposed abstract execution model for a shared memory multiprocessor system (Sequent Balance). The new extension to OSC indeed enabled us to exploit dynamic parallelism, particularly found in the *divide-and-conquer* style algorithm.

Our attempt to extend the applicability of Sisal lead to the development of compilation schemes of Sisal for SIMD multiprocessors. Transformation schemes of data-flow graph (IF1) to data-parallel programming paradigm have been proposed. In addition, this thesis presents quantitative and qualitative comparisons

between MIMD and SIMD multiprocessor in terms of both performance and the programmability.

Our recent research has been focused on the network parallel computing to take advantage of its cost-effectiveness. Moreover, the assumptions of the proposed *decoupled abstract execution model* well match to the constraints raised in the network parallel computing. In the later part of the thesis, a runtime system and the allocation schemes for a network of workstations are presented.

Chapter 1

Introduction

It is generally understood that the device technology will soon reach to its limit, and the architectural solution could only satisfy the demands which arise in high performance computing area. Parallel computing is thus becoming a major trend in solving many scientific and symbolic problems. However, despite the common understanding that the parallel computing can really improve the performance of many complex problems, the *practice of parallel programming* is still far from being an affordable solution for the application developers due to exponentially increased complexity of programming compared to the traditional sequential programming. This chapter describes the issues with parallel computing/programming and also the objectives of my research.

1.1 Motivation

Many multiprocessor systems have been introduced, and some of them are commercially available. However, programming multiprocessor systems is far more complex than conventional sequential systems for the following reasons:

- **Indeterminacy**: The order of events during the execution of a parallel program may become almost indeterminate. The programmer has to pay very close attention to the synchronization of concurrent activities in order to get the desired results. Moreover, debugging schemes based on a sequential model cannot be applied to parallel models.

- **Optimization:** In addition to the traditional optimization schemes embedded in very sophisticated modern compilers, further optimizations should be performed either by the programmer or by the compiler. Decomposing the program into several components (partitioning) and the mapping each component to an execution unit (allocation) are indeed crucial issues for the efficient execution of parallel programs. Moreover, theses optimizations are heavily dependent on individual hardware configurations. Therefore, we may need to apply different optimization criteria for different types of machines in order to get the best performance for a given parallel program. Sometimes, optimization should be done at the algorithmic level, and this means we may have to rewrite the whole program if an application must be implemented from one machine to another.

While the above issues are also raised to some extent in a sequential programming environment, they are overwhelming in parallel programming environments.

Programming environments of existing multiprocessor systems are mostly based on traditional programming languages with their own extensions for parallel execution such as new parallel constructs and/or parallel library functions (for example, MPL [53], Sequent-C [59], C* [4]). There are also several languages in which the notion of parallelism is built into (for example, ADA [28], OCCAM [40]). Some experiment shows however that due to the different ways of expressing parallelism in each system, the different viewpoints must therefore be assumed when programming each system [49].

In a pure functional programming language [8], parallelism need not be expressed by the programmer since it is implicit. Because a program written in a pure functional language is side-effect-free, the relative order of the program statements is not as significant as in an imperative language. Each statement can be executed in any order as long as the data dependencies are respected. Therefore, it is easier for the compiler to extract parallelism out of a program written in a functional programming language. Moreover, programs can be easily ported to other platforms since details about specific machines or operating systems can be hidden from the programmer.

However, a functional language alone will not guarantee ideal parallel performance. Many algorithm design, high-level programming and compilation issues remain to be investigated. For instance, a poor choice of algorithm which does not have sufficient parallelism, a weak compiler which cannot exploit the parallelism properly will severely reduce the overall performance that can be delivered by parallel implementation.

There have been many architectural solutions for the efficient execution of functional programs by incorporating the semantics of the functional program into hardware design. The data-flow architectures [22, 42, 6] and their successors in a hybrid form of data-flow and von Neumann architectures [48, 57, 32] have been introduced to provide the functional semantics built into their hardware. However, implicit parallelism through functional programs can indeed be exploited on conventional architectures. OCCAMFLOW [39], OSC [14] and TAM [21] are the projects which pursued the non-architectural solution to the parallel execution of functional programs. OCCAMFLOW have demonstrated that SISAL [55] programs can be implemented on a distributed memory multiprocessor systems, specifically on a network of Inmos Transputers™, and OSC have been successful in running SISAL programs on a wide variety of shared memory multiprocessor systems. While the above two projects have exploited parallelism at task or process level (*coarse grain*), TAM proposed abstract model for *fine-grained* parallelism with no, or minimal hardware support. Apparently, the major advantage of these projects is that we do not need any special hardware support for the parallel execution of functional programs, and we can therefore take advantages of functional semantics on existing parallel machines for the better programmability.

Motivated by all these works, I began investigating a framework in which we can describe and encode the problem in a functional language and execute it on a wide variety of platforms. For shared memory multiprocessor (SMM) systems, OSC did a reasonable job, but its applicability was limited to exploit parallelism only for *do all* (or *for all*) loop. Moreover, for distributed memory multiprocessor (DMM) systems and SIMD (single instruction stream multiple data stream) type of architecture, OSC could not deliver any performance improvement. However, OSC was a good starting point for my research and it indeed provided me

many interesting ideas such as worker-based runtime system. I first started to extend the capability of OSC to exploit parallelism for general function call. [71] And then I figured that the *function call* can be extended to a general means for parallel execution at coarse-grain level. The appropriate model for this coarse-grain parallel execution has been thus defined and the corresponding optimization schemes have been developed. It turned out that my approach was indeed very effective on the network of workstations as well as conventional SMMs. [70]

1.2 Research Objectives

In the prehistoric era of computers, when machines can understand only the commands in 0's and 1's, the main job of programmer is to translate the human commands to the sequence of 0's and 1's to communicate with machines. Soon after, the mnemonic notation for the binary stream has been used by the introduction of *assembler*. And then mathematicians developed higher level programming languages together with compilers for better notation of the problems to solve with computers. After generations of programming languages and compilers, we now can describe fairly complex problems in more human-friendly programming languages and compilers can perform very complex optimizations for the target platform. Writing a program is no longer a job for computer engineers or scientists, rather it is considered a way of solving problem for many scientists and engineers in the specific application fields.

However, introduction of multiprocessor systems raised a very fundamental question:

> "Can you solve this problem faster with this brand new multiprocessor system ?"

Ideally speaking, the answer is always 'yes'. The more correct question should be:

> "How can you solve this problem faster with this brand new multiprocessor system ?"

Now we have a real tough question. In order to answer for this question, one must explain how things work in parallel in a multiprocessor system, what kind of

algorithms should be applied, and many other technical details specific to the given platform. As we already have discussed in the previous section, there are many reasons that the nominal programmer cannot easily adopt parallel programming. Therefore we are facing a similar situation as when the first digital computer was introduced.

The ultimate goal of this research is thus *to provide an environment for general-purpose parallel computing.* In this environment, programmers no longer have to customize programs to "fit" the underlying machine architecture in order to get good performance. There are tons of research issues to achieve this goal. As discussed in the previous section, many architectural and software solutions have emerged and many of them were indeed successful at least within the scopes of their projects. With this ultimate goal in mind, we have the following research objective:

> *To provide a framework for the parallel execution of functional programs on a wide variety of platforms.*

There are two sub-goals which comprise this research:

- To define an appropriate execution model and to develop some optimization schemes for this execution model.
- To develop runtime system for various types of platforms for the application of proposed execution model.

Our approach is to exploit *coarse grain* parallelism by extending the semantics of the sequential function call to the parallel function call. The coarse-grain approach is well suited in conventional multiprocessor systems where the synchronization cost is a significant factor in the overall cost of executing a program. Especially in the network parallel computing (chapter 6), creating a task on a different host and/or message passing between hosts can easily dominate the overall performance of a program. Therefore each *grain*, *i.e.* the unit of parallel computation, must be reasonably big enough compared to the synchronization overhead to achieve speedup by parallel execution. A function – whether it is a user defined one or a compiler generated one during the process of program decomposition – is thus a good choice for our coarse-grain approach. The other advantage of taking

a function as a smallest parallel execution unit is that in functional programs, where there is no concept of state, all the input and output to/from a function is clearly defined as function arguments. This characteristics is particularly important for distributed-memory multiprocessor systems since message passing takes place only for the arguments/results of functions. *Function call* and the *return* mechanism can therefore be greatly simplified without any side effects which could have never been possible for a program written in an imperative language.

1.3 Outline of the Dissertation

This thesis consists of seven chapters. Chapter 1 includes motivation and the research objectives. In chapter 2, several previous and on-going researches which have directly or indirectly affected this research are summarized.

Chapter 3 contains the description of our abstract execution model and other related issues. *The decoupled execution model* is defined particularly for the exploitation of medium-to-coarse grain parallelism based on our approach. Several issues such as scheduling, partitioning and the optimization of control-flow is also described in the chapter.

Chapter 4 presents the extension to the Sisal compiler to enhance the applicability of OSC (section 2.2.3). This demonstrates that the *parallel function call* can be effective for the exploitation of coarse grain parallelism, particularly *dynamic parallelism* incurred by dyadic recursions. A general purpose shared memory multiprocessor system has been chosen as a target architecture.

Chapter 5 describes Sisal programming on an SIMD multiprocessor system. Schemes to transform IF1 graph (section 2.2.2) for data-parallel program is also presented. Unlike MIMD multiprocessors, dynamic-parallelism cannot be effectively exploited on a data-parallel machines. A different algorithm therefore has to be employed for SIMD multiprocessors. At the end of the chapter, both qualitative and quantitative comparisons between MIMD (particularly shared memory multiprocessors) and SIMD machine is presented.

In chapter 6, I address the parallel computing on the network of workstations. Network-based parallel computing is becoming a popular solution for high performance computation due to its flexibility and, most of all, the cost-effectiveness. Moreover, the abstract execution model described in chapter 3 particularly well fit to the network parallel computing. In this chapter, I also describe a worker-based runtime system based on PVM (Parallel Virtual Machine) together with related issues pertinent to heterogeneous network parallel computing.

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

Chapter 2

Background

This chapter describes previous researches which provided some background to this research. Data-flow execution model has been the first motivation to start this research. Particularly, the hybrid data-flow architectures have provided some insights to fill the gap between the idealism of pure fine-grained data-driven execution model and the conventional parallel computing based on von Neumann type of computation model. The practical parallel programming based on the data-driven execution model has been then realized by the introduction of a few functional languages. Among these languages, Sisal has been developed specifically targeting, but not limited to, the scientific applications. OSC (Optimizing Sisal Compiler) has indeed boosted the acceptance of Sisal as a general purpose parallel programming language in the parallel computing community.

2.1 Data-flow Model of Computation

2.1.1 Data-flow Principles

The data-flow model of computation is a simple and powerful way of describing parallel computations. A data-flow program is a directed graph made of nodes (actors) connected by arcs (links). The input and output arcs can carry tokens bearing data and control values. The arcs define paths over which tokens are conveyed from one actor to another. An actor is enabled when all its input arcs carry tokens and no token exists on its output arcs. It can then fire (*i.e.*, the instruction is executed), placing tokens on the output arcs. A program can be

constructed by putting together these actors as well as conditional actors. When loops are involved, it is necessary to distinguish between the data for different iterations. Figure 2.1 shows the execution of an actor. In the left graph, the input arcs of the actor carry tokens that bear data values. After execution, only the output arcs carry data tokens. These data tokens are sent to the instructions which need them. In other words, the execution of instructions is based on the flow of data.

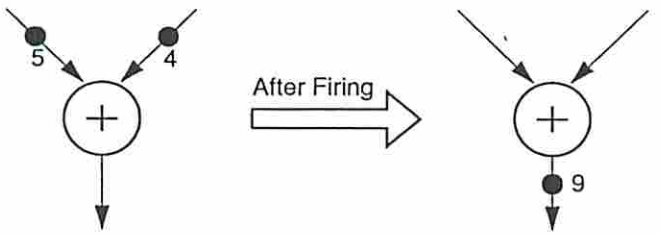


Figure 2.1: Execution of a data-flow actor.

The basic principles involved in data-flow processing can be summarized as follows:

- Data-flow programs are made of actors (instructions).
- Instructions communicate data over data arcs between data-flow actors.
- Data is transmitted in a packet called token.
- When an operation is executed, the input data values are consumed. And then the results are formatted into tokens that are placed on the output arcs of the actor.
- Operation sequencing is based on availability of data values.
- Operations are functional and do not produce side effects.

Many architectures have been designed specifically for data-flow model of computations. [5, 22, 42, 67, 46] They can be classified into two different execution model: static and dynamic. Static data-flow models allow at most one token per arc in the data-flow graph, whereas the dynamic model of data-flow allows tagged tokens and thus permits more than one token per arc. In [66], several data-flow architectures in these two categories have been compared.

2.1.2 Hybrid Data-Flow Architectures

2.1.2.1 From Pure Data-Flow to Hybrid Architecture

The early data-flow architectures have followed the data-flow principles as faithfully as possible at the architectural level, *i.e.*, synchronization and the exploitation of parallelism were all done at architectural level. Some notable machines such as Manchester Machine [42, 68] and SIGMA-1 [46, 45] belong to this first generation data-flow architectures. MIT Monsoon [20, 44] can be also categorized into a pure data-flow machine while its architecture is quite different in terms of *token matching* mechanism.

After series of researches on the pure data-flow architectures, a few points have been raised as follows:

- Implementation of too many functions in the architectural level has resulted in the overly complex hardware which is economically infeasible. Particularly, the dynamic data-flow architecture where multiple tokens can exist on a single arc requires a special hardware for synchronization (matching unit). This matching unit can be a bottleneck in the circular pipeline of the data-flow processor as well as it is very expensive to implement. The ETS (External Token Store) mechanism of Monsoon [20] is one solution which relies more on compile-time analysis for a simpler synchronization mechanism to reduce hardware cost and matching overhead.

- Dynamic scheduling at the instruction level does not guarantee the effective exploitation of parallelism. This fine-grained scheduling is indeed excessive in many cases and thus degrades the overall system performance. Instead, by the analysis of data dependency at compile-time, any closely dependent instructions can be grouped together and statically scheduled without losing too much parallelism. In this case, many scalar optimization techniques for von Neumann processor can also be applied. For example, in the data-flow graph shown in figure 2.2, instructions 1, 2 and 3 have data dependency and thus cannot be executed in parallel. By the same token, instruction 5 is dependent on instruction 4. Therefore once instructions 1 and 4 are

dynamically scheduled for parallel execution, rest of the instructions, *i.e.*, instructions 2, 3 and 5 can be statically scheduled within each shaded group.

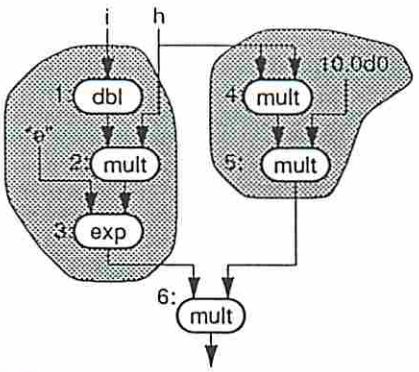


Figure 2.2: A portion of a data-flow graph: Group of nodes can be combined into a bigger node and thus scheduled statically within the bigger node.

In order to overcome the problems pointed in the early pure data-flow architectures, the next generation data-flow architectures have gradually become hybrids. With hybrid architectures, the fine-grained parallelism is exploited according to the data-driven model while a stream of sequential instructions are executed according to the von Neumann execution model. Moreover, as can be seen in ETS, some functions which had been implemented at the architectural level have become a part of compiler or runtime system at the software level. It has become also possible to employ *off-the-shelf* processors for a new hybrid architecture.

2.1.2.2 Some Hybrid Architectures

There are some hybrid architectures which have directly (or indirectly) affected the definition of *decoupled abstract execution model* (chapter 3) which is a base execution model of this research.

P-RISC

The main objective of P-RISC (Parallel-RISC) is to develop an architecture for parallel processing which can exploit fine-grain parallelism in a data-driven execution model out of a von Neumann type of architecture [57]. The main idea of P-RISC is that some functions in a pure data-flow architecture which

were transparent to the compiler, such as transmission of output tokens to their respective destination, token matching, creating a thread or new activation, and receiving tokens are now under explicit control of the compiler. There are four instructions to handle these functions:

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

The `fork IPt` instruction creates two active threads by queuing two *continuations* `<FP.IP+1>` and `<FP.IPt>`[1] in the token queue where `FP` and `IP` is a current *frame pointer* and a *instruction pointer* respectively. 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.[2]

While `fork` is used to create a new thread in the same activation 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 first one is a command and the rest in the message are operands to the command. The second operand `[FP+c]` is the descriptor of the thread to be activated. Upon arriving at the destination, the value at the first operand is stored in the frame memory location whose offset is indicated by the third 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. A remote read request is implemented by the message of the form:

[1] *i.e.*, `<FP.IP+1>` and `<FP.IPt>` are the starting points of the two new threads.
[2] This implies that for all `joins`, two inputs are expected.

<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 in 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. After sending the read request, a P-RISC processor executes other active thread from the token queue after dispatching. When a READ message is received, the memory handler saves the thread descriptor and 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.

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.

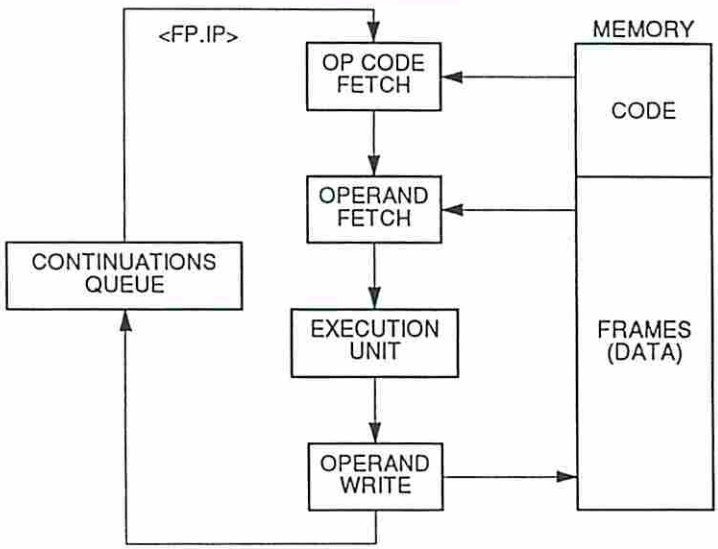


Figure 2.3: Organization of the P-RISC processor.

A processor named *T (pronounced "start") is being developed based on the concept of P-RISC. The *T processor is a extended version of the existing Motorola 88110 superscalar RISC processor [10, 23, 58] where MSU (Message and Synchronization Unit) has been added to the existing function units. The main functions of the MSU are handling messages and scheduling threads. For network

support, MSU provides a set of transmit and receive registers. User-level instructions can be used to utilize these registers and thus to send and receive messages without the operating system support. The maximum length of message is fixed at 24 words. The *T processor also provides mechanisms for multithreading. Total of 64 thread descriptors can be stored in the *Microthread Stack*. There are 8 registers which are used to store *scheduled threads*. The *T tries to pick up a thread from the *Scheduled Microthread Descriptor Registers*. If it cannot find any threads from the registers, it then looks for the *network receiver* and then *the Microthread Stack*.

USC Decoupled Architecture Model:

As opposed to the pure data-flow architecture model, it has been shown that combining simple nodes into larger *macro nodes* can improve overall performance without significant loss of parallelism [37, 32]. In this type of execution model, each *macro node* can be executed in the conventional von Neumann processors with advanced pipeline techniques for the execution of the sequential streams of instructions belonging to macro nodes. The USC Decoupled Architecture Model has thus been developed to take benefits of the variable resolution macro nodes.

The basic idea of the USC Decoupled Architecture Model is to have separate processors for actual computation and the execution of the data-flow graph. Figure 2.5 shows the architecture model which consists of two units, *Computation Engine* (CE) and the Data-Flow Graph Engine (DFGE). The data-driven execution model is implemented by the DFGE and the sequential streams of instructions are executed in the CE to take advantage of the von Neumann processors, *i.e.* The DFGE performs token matching and when all input tokens of a macro node are available, the node is scheduled for execution by th CE. Two queues, the *Ready Queue* (RQ) and the *Acknowledge Queue* (AQ), are used to connect the DFGE and the CE. RQ holds the descriptors of ready nodes and the AQ holds the descriptors of the nodes that have been completed by the CE. Each of DFGE and CE has its own local memory, the *Graph Memory* and the *Computation Memory* connected to the *Graph Cache* and the *Computation Cache*, respectively. The contents of both caches are updated by corresponding *Queue and Cache Controller* (QCC).

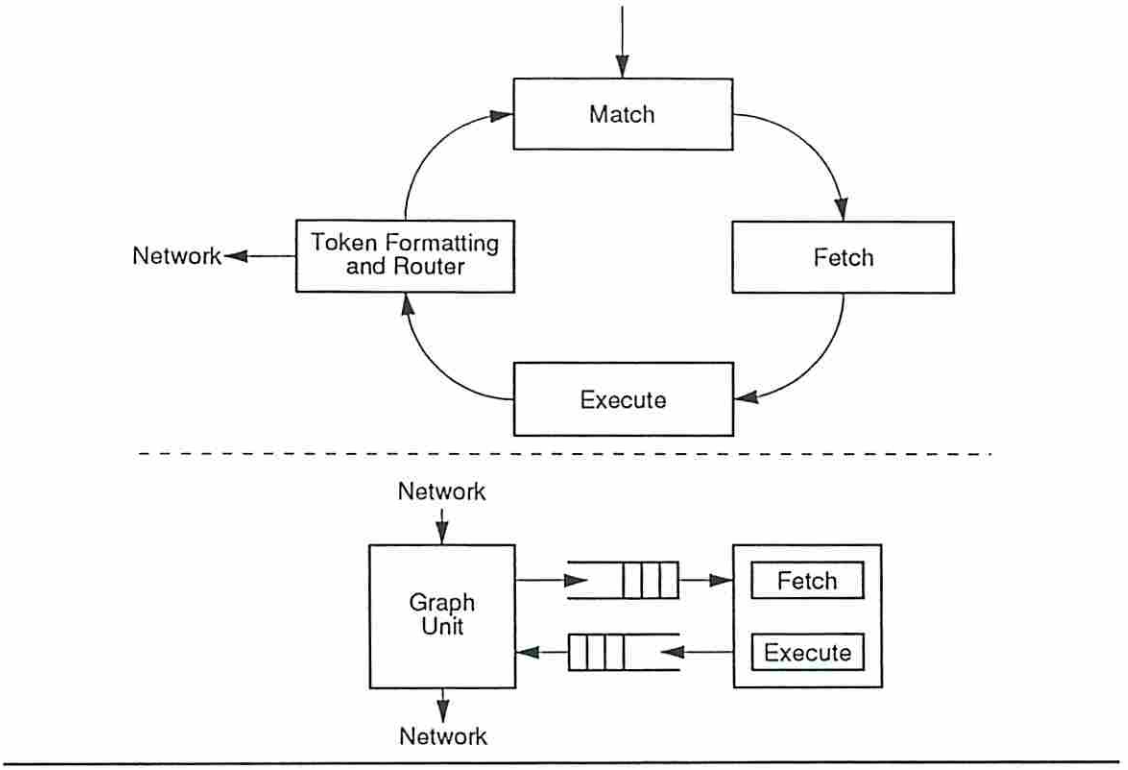


Figure 2.4: Basic data-flow architecture organized as a cyclic pipeline (upper part) and Decoupled Graph/Computation configuration (lower part).

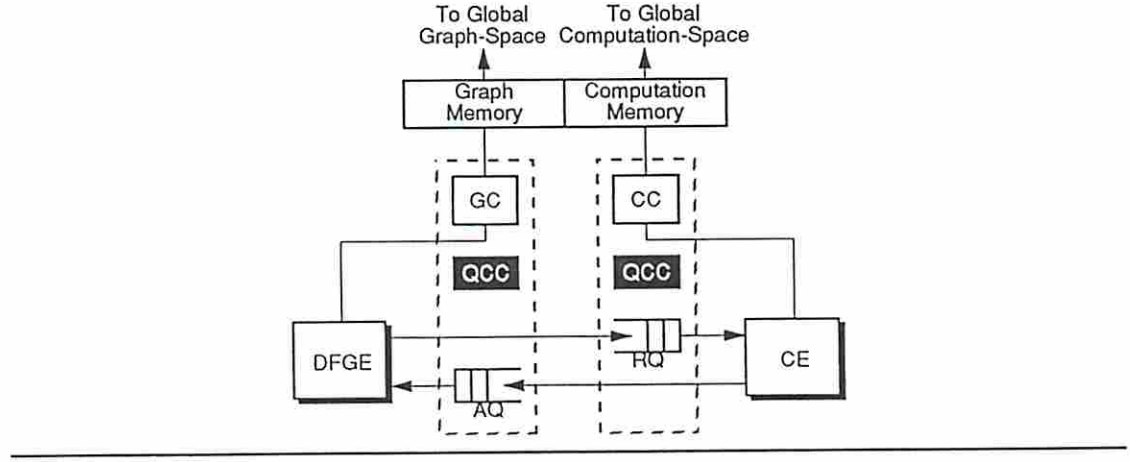


Figure 2.5: Data-flow processor with decoupled graph and computation units.

Other Hybrid Architectures

P-RISC and the USC decoupled architecture model provided the direct motivations to *decoupled abstract execution model* which will be described in chapter 3. There are other hybrid architectures which have not been described here such as EM-4 of ETL [64], the McGill Dataflow Architecture Model (MDFA) [35], the Hybrid Architecture [48] and the Tera machine [1].

The more extensive survey of data-flow architectures can be found in [38].

2.2 The Parallel Programming Environment

2.2.1 Overview

In general, parallel programming can be summarized in the following steps:

- *Designing or finding an algorithm for the application:* The first step is to find an appropriate parallel algorithm to solve the application. It is sometimes necessary to design a new algorithm. In this latter case, a verification of the algorithm will also have to be performed.

- *Initial coding of the algorithm:* Once the right algorithm has been selected, it should be coded and checked for functional correctness. Any programming language can be used for this purpose. However, one may have to select a language which is supported by the programming environment (SISAL in our case).

- *Justification of parallel implementation:* For a parallel implementation of the algorithm, one must ensure that the algorithm has sufficient inherent parallelism. If the algorithm itself does not have any parallelism, any further painstaking effort for implementation and optimization on the target parallel machine will be for naught. Therefore, before undertaking a serious implementation, justification for the parallel execution should be accomplished.

- *Implementing and optimizing for the target platform:* This step is truly machine-dependent, and requires a good knowledge of the target machine,

including features geared to parallel execution. In the ideal case, the stock compilers will perform automatic optimization for parallel execution of the program, while the programmers need only translate the algorithm directly to the target language. However, the optimization, as discussed earlier, includes several tasks, none of them trivial. There is no general optimization scheme. In fact, the schemes will differ from machine to machine, and even from application to application. Therefore, in most cases, the programmers have the responsibility to perform at least part of the optimization manually. Moreover, although an algorithm may possess sufficient inherent parallelism, it may not be possible to efficiently implement it on a wide range of architectures due to the different features presented by each platform (*e.g.*, shared memory vs. distributed memory). Therefore, it is often the case that the algorithm must be redesigned for a different machine. Indeed, it is desirable to have the target machine in mind from the algorithm designing step in order to minimize the risk of poor match of the algorithm to the target machine.

- *Measuring and analyzing the performance:* In the final step, the performance is measured and analyzed. The results may be used for further improvements of the application.

Our parallel programming environment is based on the SISAL language [55, 71]. The front end of the SISAL compiler generates a data-flow graph, called IF1 [65]. Further analysis is performed on this IF1 graph to verify the correctness and the potential parallelism of the program using DI (Debugger and Interpreter).

Once we have verified that our implementation of the application has enough potential parallelism, we run the program on the target machine by using OSC. The overall programming environment is shown in Figure 2.6.

The first phase (the upper block) corresponds to the optimization at the algorithm level, and in the second phase (the middle part and the lower block) we perform implementation on the target machine. Each component of the programming environment is further discussed in the following sections.

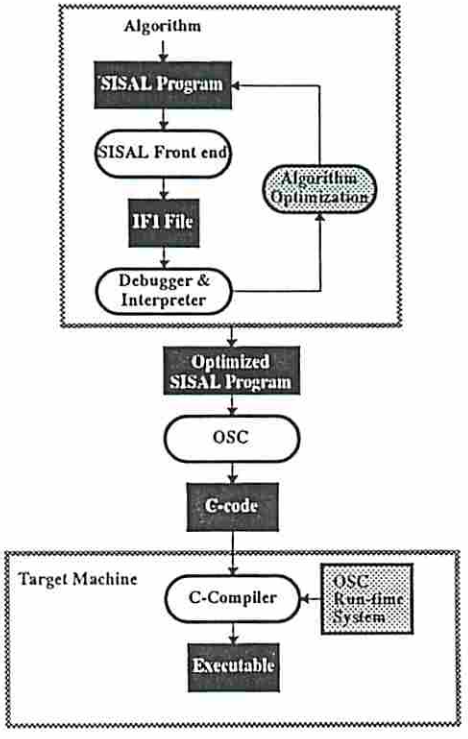


Figure 2.6: The overall programming process

2.2.2 SISAL and IF1

SISAL (Streams and Iterations in a Single Assignment Language) [55] is the functional language which we chose for parallel implementation of our application. SISAL has been proven very effective in developing applications for multiprocessor systems. The optimizing SISAL compiler (OSC) has been developed for various shared memory multiprocessor systems [14, 17, 62]. It has also been shown that the performance of several SISAL programs is comparable to (or better than) that of Fortran programs [15].

SISAL has the following characteristics:

- *It is applicative*: Every function call returns one or more values. There is no global space for sharing the values. Communication between the caller and the callee function is done only through the input arguments and the returned values.

- *It follows the single assignment rule*: Each variable can be assigned a value only once within the same scope. (although it can be used many times in the life of the program.) In the single assignment rule, there is no aliasing problem, thus facilitating the detection of parallelism in a program. [3]

- *It is strongly typed*: Each value has a type. Thus, in the definition part of each function, every argument is given a correct type.

There are six basic scalar types in SISAL: boolean, integer, real, double real, null and character. A data structure in SISAL consists of arrays, streams, records and unions. Each basic data type has its associated set of operations, while record, union, array and stream types are treated as mathematical sets of values just as the basic scalar types. Interestingly, there is a special value *error*. This value is associated with a special operator *iserror* which is used to handle exceptions at runtime.

Three types of control mechanisms are implemented in SISAL: *forall*, *select*, and the *iteration* constructs. These constructs can be nested and are implemented

[3] Consider conversely the *common* or the *equivalent* statements in Fortran. They allow reference to the same storage cell under many different names.

by multilevel (hierarchical) graph structures in the lower-level graph representation (IF1) . The *forall* construct is used to specify parallel loops enabling parallel execution of the loop-body while the *iteration* only allows sequential execution of the loop body. *Select* corresponds to an *if–then–else* structure. Examples of SISAL programs are shown in figure 2.7.

```
% Example of forall
function DotProd(n:integer;a,b:array[integer] returns integer)
        for i in 1,n
                s := a[i]*b[i]
        returns
                value of sum s
        end for
end function

% Example of iteration
function factorial(n:integer returns integer)
        for initial
                f := 1;
                i := 0;
        while i < n
        repeat
                i := old i + 1;
                f := old f * i;
        returns
                value of f
        end for
end function

% Example of select
function Max(a,b:real returns real)
        if a > b then a else b end if
end function
```

Figure 2.7: Examples of SISAL program

The IF1 (Intermediate Form 1) graph is produced by the SISAL compiler. Basic components of an IF1 graph are *graph*, *node*, and *edge (arc)*.

There are two types of IF1 nodes: the simple node, and the compound node. While simple nodes implement actual operations, compound nodes correspond to

the control structure of the SISAL program. Each compound node has one or more subgraphs according to the node type. For example, the *forall* compound node has three subgraphs, *generator*, *body*, and *returns*. Each subgraph of a compound node is controlled by predefined implicit data dependencies. In other words, no explicit arcs exist between subgraphs. There are four types of compound nodes: *forall*, *select*, *iteration*[4], and *tagcase*.

Each edge is associated with a type. This implies that each edge can carry either a single value (scalar) or structured data (array or record) together with the source node (and port) and the destination node (and port). When an edge also has immediate data, it is called a *literal edge* and has no source node.

As an example, the IF1 graph of the `DotProd()` function (the SISAL code is shown in figure 2.7) which calculates the *inner-product* of two vectors is shown in figure 2.8.

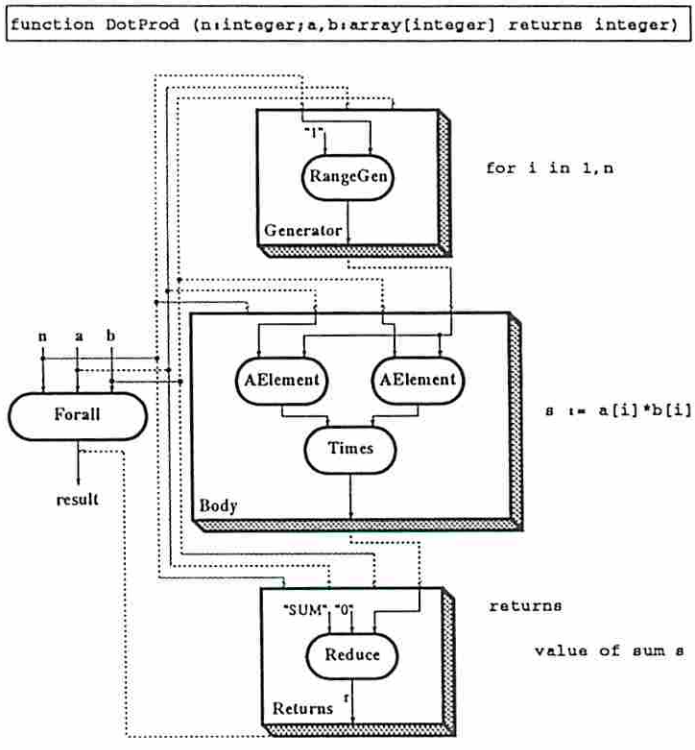


Figure 2.8: IF1 graph of dot-product function

[4] *Iteration* is actually a combined construct of two types of sequential loops (*loopA* and *loopB*)

In the figure, the dotted lines are not real edges in the IF1 graph. They are only defined by implicit dependencies between the subgraphs of the *forall* compound node.

2.2.3 OSC

2.2.3.1 Overview of the OSC

OSC (Optimizing SISAL Compiler) has been developed as a part of a SISAL project [33]. The project's objectives are to:

1. define a general-purpose applicative language,
2. define a language-independent intermediate form for data-flow graphs,
3. develop optimization techniques for high-performance parallel applicative computing,
4. develop a micro-tasking environment that supports data-flow on conventional computer systems,
5. achieve execution performance comparable to that of Fortran, and
6. validate the applicative style of programming for large-scale scientific applications.

OSC targets a wide variety of platforms including conventional single processor machines and multiprocessor systems. The version we used for our study (V12.7) has been ported on most single processor machine running UNIX™ and the following platforms:

1. Sequent Balance running DYNIX
2. Alliant FX series running Concentrix
3. Encore Multimax running Umax
4. Sequent Symmetry running DYNIX
5. Cray Y-MP or X-MP running UNICOS
6. Cray 2 running UNICOS
7. SGI running IRIX

The portability of OSC is due to the choice of C as a target object language. Most of the compilation steps are therefore the same on all the platforms. The machine dependent features of each platform are implemented in the runtime library of OSC. The structure of OSC is shown in figure 2.9.

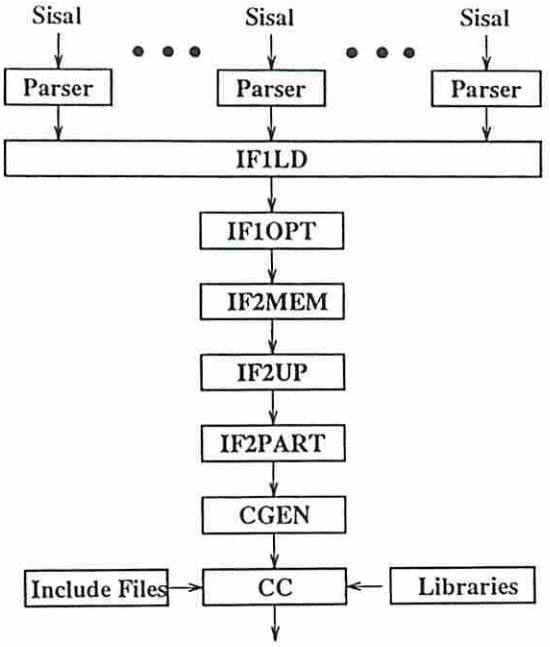


Figure 2.9: Structure of OSC

2.2.3.2 Front end

A SISAL program can be written in several modules. Each module is separately compiled to IF1, and IF1LD then combines separate IF1 modules to produce a monolithic IF1 graph.

2.2.3.3 Optimization

IF1OPT performs the following machine-independent optimizations:

1. Function in-lining.
2. Record and array fission.
3. Loop invariant removal.
4. Common subexpression elimination.
5. Dependent and independent fusion.
6. Loop unrolling.
7. Constant folding and operator strength reduction.
8. Dead code elimination.

The major overhead of running a SISAL program (and any pure functional program) is caused by copy operations of the structure data because of its *side-effect-free* principle. OSC reduces this overhead by incorporating *in-place* operations (also called *AT* operations) which are used to access structure data by reference rather than by their values while preserving the overall data dependence. In figure 2.9, IF2MEM and IF2UP extends the IF1 graph with these *in-place* operations. The extended graph is called IF2 [69]. The main job of IF2MEM is to preallocate storage for new structures (*build-in-place* analysis) while IF2UP facilitates the modification of a single data inside the storage (*update-in-place* analysis). Note that it is necessary to serialize some structure handling operations in order for the IF2 graph to work correctly after *update-in-place* analysis. However it should also be noted that this loss of parallelism is compensated by the reduced overhead of creating/copying whole structures whenever their components are modified.

IF2PART is a parallelizer designed to define the desired granularity of parallelism. The analysis is based on the estimation of execution time which are determined by various parameters such as computation time, and spawning overhead. The current implementation selects only *forall* compound nodes to slice the loop into many pieces for parallel execution. The more detailed partitioning scheme is described in [62].

CGEN translates the optimized IF2 graphs into equivalent C code. The generated C code together with the machine-dependent runtime libraries are then compiled to produce an executable. The OSC runtime system is described in the next section.

2.2.3.4 OSC Runtime System for Parallel Execution

The runtime system supports the parallel execution of SISAL programs, provides general-purpose dynamic storage allocation, implements operations on major data structures, and interfaces with the operating system for input/output and command line processing. In this section, we mainly describe how this OSC runtime system supports the parallel execution of SISAL programs.

The OSC runtime model defines its own *activation record* for each concurrent task (similar to a *process control block* in traditional operating systems) and each task is executed by one of the worker processes. On each processor, a *worker process* is running concurrently, waiting for any task to appear. There can be a global *ready queue* for all the worker processes, or each worker process can have its own *ready queue* from which it dispatches the next task to do. Upon encountering the code segment[5] which is to be executed in parallel, an activation record is created for the task corresponding to the code segment, and then queued in the global ready queue[6]. After placing a subtask into the ready queue, the parent task continues its execution. When the results are needed, the parent task should wait for the subtask. While waiting for its subtask, the parent task is suspended and the worker which was running the parent task searches for any other task to execute, and thus increases the utilization of each worker process.

Note that there is no single centralized scheduler in the OSC runtime model. Each idling worker process keeps trying to dispatch a task. Further, any worker process can activate a new concurrent task during the execution of any task.

The mechanism of the OSC runtime model can be briefly described as follows:

- In the beginning, n worker processes are created on n processors. Only worker 0 continues the execution of the SISAL program while the others are waiting for new tasks.

[5] Indeed, the code segment is converted into a function by OSC when it is to be executed in parallel.

[6] Alternatively, if we use separate queues for each worker process, a task is placed into the proper queue according to the allocation policy. Hereafter, for simplicity, the existence of a global ready queue will be assumed.

- Upon encountering the code (function) that should be executed in parallel, the current task builds the activation record for a new subtask, puts it in the ready queue, and then continues its execution.
- When the result of the function (which was spawned as a subtask) is needed, repeat the following:
  1. If the subtask has completed, return to the parent task.
  2. If the subtask has not completed, suspend itself and check whether any other tasks can be executed.
- When the execution of the program is completed, shut down the run-time system and output results. Note that worker 0 is the only process which can start the shut-down procedure for the normal exit of the program.

The current OSC implementation looks only at *forall* loops for parallelization. The loop-body is turned into a C function by the C-code generator of OSC. Further the m slices of the same loop-body are created with different index arguments. A worker, after initiation of m slices, immediately jumps to the waiting loop and suspends the current task to execute other tasks if any.

Chapter 3

Decoupled Execution Model

This chapter describes an execution model which is appropriate for the exploitation of medium-to-coarse grain parallelism. The basic idea is to separate the actual program code from the control-flow. This work is indeed inspired by the two hybrid data-flow architectures, the USC decoupled architecture model [32] and P-RISC [57]. The USC decoupled architecture model has provided a basis for the *larger grain* parallel execution model while the essential primitives and their semantics for our abstract execution model were motivated by the P-RISC instructions. The analysis and the optimization on the control-flow is also described in the chapter.

3.1 The Model

3.1.1 The Abstract Execution Model

For the parallel execution of any programs, we must identify the usable parallelism. However, the analysis of the program in the instruction (or statement) level is too complex and is often found that the analysis itself is impossible due to its complexity. By writing a program in functional programming language, the complexity of analysis can be greatly reduced. In principle, there is indeed no need for extracting parallelism in functional programs since the fine grained parallelism is already implicitly defined by the program itself.

Once parallelism is identified, what we need for performance enhancement is how to control the uncovered parallelism for a given target machine. Since we are

targeting a wide variety of platforms, we need a very simple but effective means for exploiting parallelism. We thus chose to use the following two primitives as our basic tools for exploiting parallelism:

- *fork* : Create a new task for a specified function and execute it in parallel.
- *join* : Wait until the specified function (task) is done.

The above two primitives are well defined in a general sense, and thus its implementation can be fairly straightforward in most multiprocessor systems. The only problem with these two primitives is that in many cases the usage of *fork* and *join*, especially *fork* can cause a big system overhead. Therefore it is desired to pay good attention to the usage of these primitives, otherwise, the parallel execution might lead to performance degradation.

We introduce an abstract execution model where the actual program code and the control of the program execution are separated. As shown in figure 3.1, each user program is divided into two parts, *control schema* and *program code*. In *program code*, all the user (or compiler generated) functions are implemented while, in *control schema*, only one of the following statements are allowed:

- *fork*
- *join*
- *a function call*

In other words, the exploitation of parallelism is *explicitly* defined in control schema. The overall performance of the program can directly be affected by this control schema and we thus need a scheme for optimization of the control schema which will be described later.

3.1.2 Control Schema and Allocation

One of the functions that can be defined in the control schema is allocation strategy. We can add extra information on the *fork* statement for allocation. Consider the following example:

```
fork f1 (...) on PE4
```

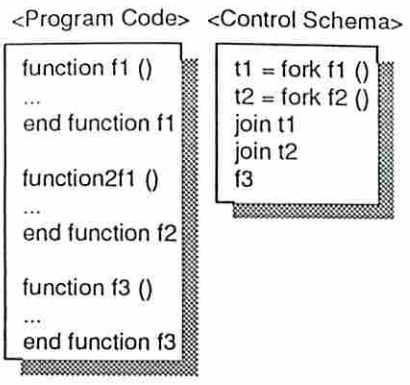


Figure 3.1: The overview of Decoupled Execution Model

In the above example, one can specify on which processor the function `f1` should be executed. This shows an example of how a *static scheduling* scheme can be implemented coupled with *control schema*. Take a look at another example:

```
fork f1 (...) on schemeX
```

The above example can be read as:

"create a new task for function `f1` and allocate this new task on a processor which is determined by some *dynamic* scheduling scheme called `schemeX`"

Any dynamic runtime scheduling schemes can be used and called from *control schema*. The usage of control schema is indeed very flexible. Not only the allocation information but also any other information pertaining to the actual execution of the function can be specified in the control schema.

3.1.3 Compilation for Control Schema

In order to incorporate control schema, the compiler must be redesigned. For example, the Sisal compiler based on the control schema is shown in figure 3.2. From a Sisal program, we obtain IF1/IF2 graph and the scalar optimization and basic parallelization (such as partitioning) is performed to generate optimized/extended IF1/IF2 graph. From this extended IF1/IF2 graph we can directly extract program code which is simply a definition of each function. The other path to the

right in the figure shows the extraction of control schema. Once the program code and control schema is obtained the final target code is generated based on them.

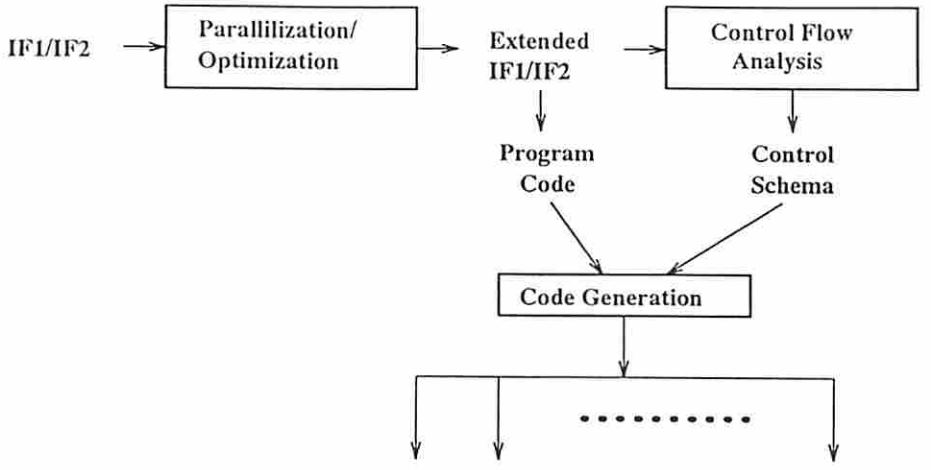


Figure 3.2: An overview of the new Sisal compiler based on the control schema

3.2 Optimization

In order to achieve performance enhancement, we mainly concentrate on the optimization of the control schema. In this section we will describe how the coarse-grain parallelism can be effectively executed by introducing some optimization scheme. Let us begin with some example written in a functional language (SISAL-like) which is shown in figure 3.3, and the corresponding data-flow graph shown in figure 3.4.

```
t1 := a (v);
t2 := b (t1, v);
t3 := c (t1, v);
t4 := e (t2, t1, t3);
r  := h (g (d(t1,t2), t4), t4, f(t3));
```

Figure 3.3: A program code example written in a functional language.

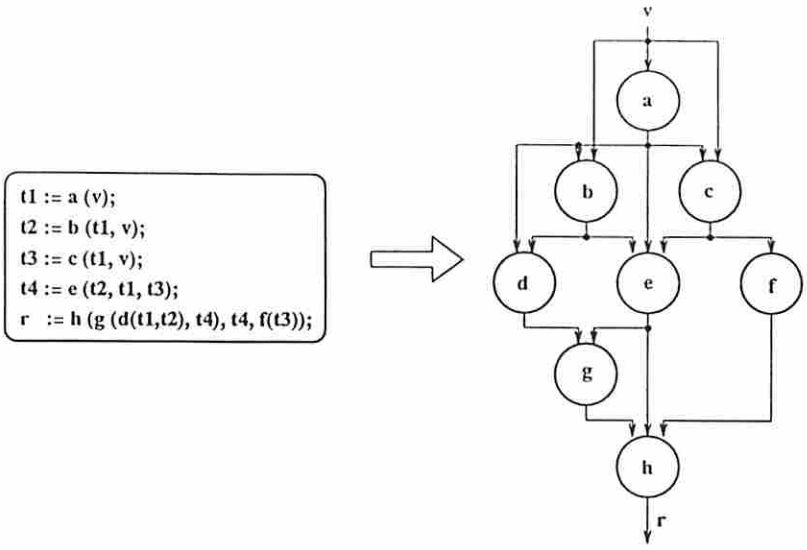


Figure 3.4: An example of functional program and its data-flow graph

The graph shown in figure 3.4 is a direct representation of the program, *i.e.* all the data paths appear in the graph to show the flow of data. Note, however, that all the edges in the graph do not necessarily imply the *direct flow of control.* For example, an edge from a to e is redundant in terms of *control-flow* since the node e is also dependent on the node b and the node b is again dependent on the node a. In the fork-join execution model, we mainly investigate the flow of the control and if there are two or more control-independent nodes, we *fork* new tasks for parallel execution of those independent nodes. When a result of *fork*ed node is needed to proceed the execution of the graph, *join* is executed to wait for it. In the rest of this section, we will describe how we derive fork-join execution model from a data-flow graph.

We first begin with the formal definition of data-flow graph.

Definition 1 *A* **data-flow graph** *is a directed graph:*

$$G = (N, E)$$

where N is a set of nodes, and E is a set of edges which correspond to the actual operations (or functions) and data.

For the graph shown in figure 3.4, N and E are as follows:

$$\begin{aligned} N &= \{\top, a, b, c, d, e, f, g, h, \bot\} \\ E &= \{\overline{\top a}, \overline{\top b}, \overline{\top c}, \overline{ab}, \overline{ac}, \overline{ad}, \overline{ae}, \overline{bd}, \overline{be}, \overline{ce}, \overline{cf}, \overline{dg}, \overline{eg}, \overline{eh}, \overline{fh}, \overline{h\bot}\} \end{aligned}$$

where $\overline{xy}$ denotes the edge from the node x to y, and $\top/\bot$ represents top/bottom of the data-flow graph.

Definition 2 *A* **dependency function**, $\mathcal{D} : N \rightarrow N^*$, *of* $G = (N, E)$ *is defined as*

$$\mathcal{D}(x) = S \text{ such that } \overline{xy} \in E \text{ for all } y \in S$$

For a given set of edges, E, the dependency function, $\mathcal{D}$, is uniquely defined and *vice versa*. Therefore a data-flow graph can also be defined by the two tuples such that $G = (N, \mathcal{D})$. The dependency function $\mathcal{D}$ of the graph in figure 3.4 is as follows:

$$\begin{aligned} \mathcal{D}(\top) &= \{a, b, c\} \\ \mathcal{D}(a) &= \{b, c, d, e\} \\ \mathcal{D}(b) &= \{d, e\} \\ \mathcal{D}(c) &= \{e, f\} \\ \mathcal{D}(d) &= \{g\} \\ \mathcal{D}(e) &= \{g, h\} \\ \mathcal{D}(f) &= \{h\} \\ \mathcal{D}(g) &= \{h\} \\ \mathcal{D}(h) &= \{\bot\} \\ \mathcal{D}(\bot) &= \emptyset \end{aligned}$$

Definition 3 *The* **dependency closure**, $\mathcal{D}^*$, *of a dependency function* $\mathcal{D}$ *is a function such that:*

$$\mathcal{D}^*(x) = \begin{cases} \emptyset & \text{if } \mathcal{D}(x) = \emptyset \\ \bigcup_{y \in \mathcal{D}(x)} \mathcal{D}^*(y) \ \cup \ \mathcal{D}(x) & \text{otherwise} \end{cases}$$

where $\mathcal{D}$ *is a dependency function of a graph* $G = (N, E)$.

Intuitively speaking, $\mathcal{D}^*(x)$ defines all the nodes which should be executed after the node x in the data-flow graph is executed. For the data-flow graph shown in figure 3.4, $\mathcal{D}^*$ is as follows.

$$\begin{aligned}
\mathcal{D}^*(\top) &= \{a, b, c, d, e, f, g, h, \bot\} \\
\mathcal{D}^*(a) &= \{b, c, d, e, f, g, h, \bot\} \\
\mathcal{D}^*(b) &= \{d, e, g, h, \bot\} \\
\mathcal{D}^*(c) &= \{e, f, g, h, \bot\} \\
\mathcal{D}^*(d) &= \{g, h, \bot\} \\
\mathcal{D}^*(e) &= \{g, h, \bot\} \\
\mathcal{D}^*(f) &= \{h, \bot\} \\
\mathcal{D}^*(g) &= \{h, \bot\} \\
\mathcal{D}^*(h) &= \{\bot\} \\
\mathcal{D}^*(\bot) &= \emptyset.
\end{aligned}$$

We call the term, $\bigcup_{y \in \mathcal{D}(x)} \mathcal{D}^*(y)$, a *derived dependency* and is denoted by $\mathcal{D}'(x)$. $\mathcal{D}'(x)$ can be interpreted as the set of nodes which can be reached via *one or more* nodes. For example, in figure 3.4:

$$\mathcal{D}'(a) = \{d, e, f, g, h, \bot\}.$$

Note that $\mathcal{D}$ and $\mathcal{D}'$ do not have to be disjoint to each other. In our example,

$$\mathcal{D}(a) \cap \mathcal{D}'(a) = \{d, e\}$$

meaning that node d and e can be reached from node a by some other paths than direct edges, $\overline{ad}$ and $\overline{ae}$.

Definition 4 *The two data-flow graphs, $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$, are* **equivalent in control-flow** *and denoted by $G_1 \overset{*}{\equiv} G_2$ if and only if*

$$N_1 = N_2 \text{ and } \mathcal{D}_1^*(x) = \mathcal{D}_2^*(x) \text{ for all } x \in N_1$$

where $\mathcal{D}_1$ and $\mathcal{D}_2$ are dependency functions of G_1 and G_2, respectively.

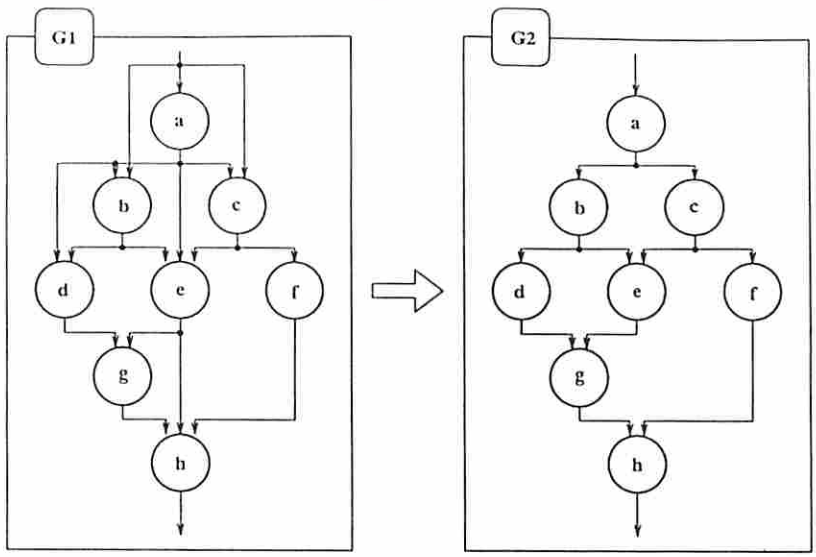


Figure 3.5: The two data-flow graphs which are equivalent in control-flow.

Consider the two graphs, G_1 and G_2 in figure 3.5. Applying the definition 3 to both graphs, the dependency closure of these two graphs are the same. This means that the any possible topological orders for both graphs are the same. This also implies that the inherent parallelism of G_1 is the same as that of G_2.

In the fork-join execution model, especially in multiprocessing environment, *fork* and *join* are usually the major overheads for overall execution time. It is therefore desirable to minimize such operations. Now we describe how we obtain the graph which yields the minimum number of *fork* and *join* operations.

Definition 5 *A data-flow graph* $G_0 = (N_0, E_0)$ *is* **minimally equivalent in control-flow** *to the graph* $G = (N, E)$ *if and only if*

$$\begin{aligned} G_0 &\overset{*}{\equiv} G \\ \mathcal{D}_0(x) &= \mathcal{D}(x) - \mathcal{D}'(x) \text{ for all } x \in N \end{aligned}$$

where $\mathcal{D}_0$ *and* $\mathcal{D}$ *are dependency functions of* G_0 *and* G*, respectively, and* $\mathcal{D}'$ *is a derived dependency of* G*.*

G_0 is indeed a graph which is equivalent in control-flow to G with the minimum number of edges. In figure 3.5, the graph G_2 is the minimally equivalent graph to G_1, since

$$
\begin{array}{lcllcl}
\mathcal{D}_1(\top) - \mathcal{D}_1'(\top) & = & \{a,b,c\} - \{b,c,d,e,f,g,h,\bot\} & = & \{a\} & = \mathcal{D}_2(\top) \\
\mathcal{D}_1(a) - \mathcal{D}_1'(a) & = & \{b,c,d,e\} - \{d,e,f,g,h,\bot\} & = & \{b,c\} & = \mathcal{D}_2(a) \\
\mathcal{D}_1(b) - \mathcal{D}_1'(b) & = & \{d,e\} - \{g,h,\bot\} & = & \{d,e\} & = \mathcal{D}_2(b) \\
\mathcal{D}_1(c) - \mathcal{D}_1'(c) & = & \{e,f\} - \{g,h,\bot\} & = & \{e,f\} & = \mathcal{D}_2(c) \\
\mathcal{D}_1(d) - \mathcal{D}_1'(d) & = & \{g\} - \{h,\bot\} & = & \{g\} & = \mathcal{D}_2(d) \\
\mathcal{D}_1(e) - \mathcal{D}_1'(e) & = & \{g,h\} - \{h,\bot\} & = & \{g\} & = \mathcal{D}_2(e) \\
\mathcal{D}_1(f) - \mathcal{D}_1'(f) & = & \{h\} - \{\bot\} & = & \{h\} & = \mathcal{D}_2(f) \\
\mathcal{D}_1(g) - \mathcal{D}_1'(g) & = & \{h\} - \{\bot\} & = & \{h\} & = \mathcal{D}_2(g) \\
\mathcal{D}_1(h) - \mathcal{D}_1'(h) & = & \{\bot\} - \emptyset & = & \{\bot\} & = \mathcal{D}_2(h) \\
\mathcal{D}_1(\bot) - \mathcal{D}_1'(\bot) & = & \emptyset - \emptyset & = & \emptyset & = \mathcal{D}_2(\bot)
\end{array}
$$

The intuition behind the minimally equivalent graph is to find the graph which has the simplest control structure while preserving the same potential parallelism as the original data-flow graph. In the next step, we generate a series of *fork*s and *join*s from this graph.

It is straightforward to extract fork-join execution model from the minimally equivalent graph. We start from the top node (denoted by $\top$) and follow the graph to generate the sequence of instructions which are either *fork* or *join* until we reach the bottom node (denoted by $\bot$). Whenever a node is ready to be executed, we just *fork* it and add it to the *active list*. When there is no more *ready nodes*, we pick a node from the *active list* and set its status as *done* (*i.e. join* it). Note that the redundant *join*s must be avoided in the generated instruction sequence while maintaining the execution order of original graph. The complete algorithm is shown in figure 3.6 and an example is shown in figure 3.7.

Assuming that the *fork* operation is very expensive compared to the sequential invocation of a node, the fork-join model can be further optimized to reduce the redundant forks and joins. If there is any consecutive *fork-join* pair to the same node, we can just replace these two instructions with the sequential call to the node. Moreover, any consecutive *forks* or *joins* can take place in any order.

Algorithm *generate_fork_join*
Input: $G = (N, E)$: A data-flow graph.
Output: A sequnece of instructions which is either *fork* or *join*.
Begin
    Find a predecessor function $\mathcal{P}$ of G such that $\mathcal{P}(x) = S$ such that $\overline{yx} \in E$ for all $y \in S$
    let $ActiveSet \leftarrow \emptyset$, and $node\_done \leftarrow \top$.
3 Doforever
        forall $n \in \mathcal{D}(node\_done)$ do
            $\mathcal{P}(n) \leftarrow \mathcal{P}(n) - \{node\_done\}$
            if $\mathcal{P}(n) = \emptyset$ then
                if $n = \perp$ then
                    return // normal return since we reached the bottom of the graph.
                endif
                generate "fork n" intruction
                $ActiveSet \leftarrow ActiveSet \cup \{n\}$
            endif
        end forall
        if $ActiveSet = \emptyset$ then
            return // no more to do (abnormal exit).
        endif
        pick a node c from $ActiveSet$ and let $ActiveSet \leftarrow ActiveSet - \{c\}$.
        generate "join c" instruction.
        let $node\_done \leftarrow c$
    End DoForever
End

Figure 3.6: An algorithm to generate the sequence of forks and joins from a data-flow graph.

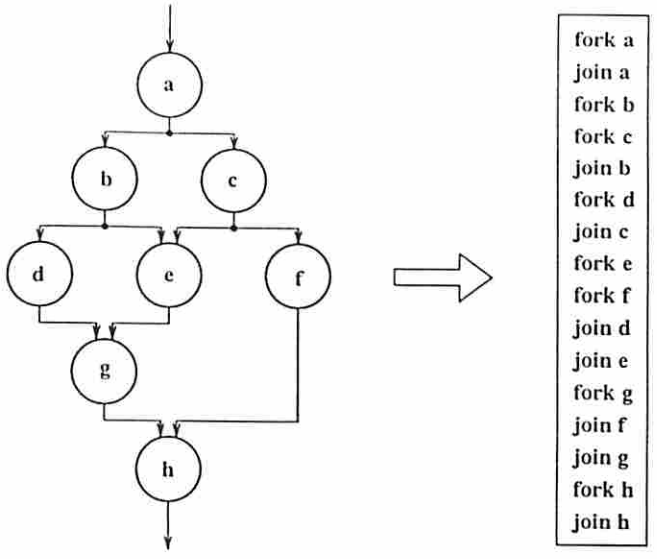


Figure 3.7: An example of converting a control-flow (data-flow) graph to *fork-join* execution model

Therefore, we can reorder the *forks* or *joins* so that there can be as many *fork-join* pairs to the same node as possible. After the reordering, we simply replace all the *fork-join* pairs with the sequential execution of the node. This optimization process is depicted in figure 3.8.

3.3 Related Issues

3.3.1 Scheduling issues in the runtime system

During the execution of a program, it is often desired to have dynamic control of parallelism. The compiler cannot extract all the parallelism statically. For example, in our previous experimentation with FFT, where the main source of parallelism is dyadic recursion, compiler cannot precisely predict the behavior of the program. Therefore, the parallel execution mainly relies on the dynamic runtime primitives (*SpawnTask*). This implies that we can easily reach to the limit of the system resources and the performance of the program can be degraded with too many outstanding parallel tasks. To prevent this overload to the system

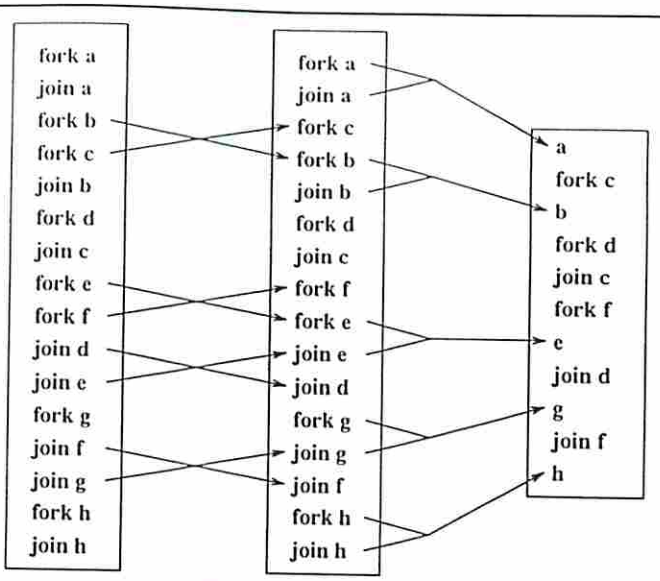


Figure 3.8: An example of removing redundant forks and joins by reordering.

resources, we need to incorporate the *discrimination function* for further parallel tasks. Consider a simple function as follows:

function *NoMoreSpawning()*
    if *number of outstanding tasks* $\geq$ *total number of processors* then
    true
    else false
    endif
end function

Now, we can insert the above function in *SpawnTask* (or *SpawnTaskGroup*) as follows:

function *SpawnTask(code, input arguments...)*
    if *NoMoreSpawning()* then
        Execute the *code* sequentially.
        returns **null** as a *task id* and exit from this *SpawnTask* call.
    endif
    Continues with the normal spawn procedure.
end function

The *JoinTask* has to be modified accordingly:

function *JoinTask(task_id)*
    if *task_id* = null then
        Immediately exits from this *JoinTask* call.
    endif
    Continues with the normal join procedure.
end function

For example:

function *NoMoreSpawning()*
    if *number of outstanding tasks* $\geq$ *total number of processors* $\times$ 2
    then true
    else false
    endif
end function

Indeed, the above function is over-simplified and we do not believe it is the optimal function for the target machine (Sequent Balance). Our major concern was to maintain sufficient number of tasks for better *utilization* of processors while keeping the system from being overloaded to prevent the excessive overhead incurred by *spawning* tasks (*parallel function calls*).

3.3.2 Partitioning issues for coarse grain parallelism

One of the most important steps in compile-time analysis for parallel execution is to identify the *useful parallelism*. A data-flow graph has the maximum potential parallelism at its finest grain (*i.e* at the instruction level). However, on a conventional machine with the aforementioned *fork-join* execution model, the exploitation of instruction-level parallelism on different processors incurs too much synchronization overhead and the performance is thus degraded. Therefore, in this research, we opt for coarse grain parallelism.

The granularity is indeed determined by the various parameters of the target system, such as number of processors and inherent runtime overhead for creating

parallel tasks. Communication cost and/or the cache performance are also very important factors which affect the granularity. All these parameters are quite different from machine to machine, and even worse, some parameters cannot be quantified. As a result, it is nearly impossible to find the universal scheme for optimal granularity. There have been many researches with regards to partitioning and allocation problem [47, 36, 61, 62, 63]. While these researches have been successful within their own architectural domain, most of them have actually failed to provide a practical scheme for general purpose conventional parallel machines. At this stage of our research, detailed partitioning schemes are yet to be investigated. Therefore, in this section we will present some issues that must be considered in developing partitioning schemes.

3.3.3 Convexity constraint vs. hierarchical partitioning

The Convexity constraint ensures that each partition can run to completion once all its inputs are available [61]. In other words, a partition cannot suspend during its execution. The main advantage of the convexity constraint is the fact that the scheduling overhead for the execution of the entire partition is introduced only once in the beginning of the execution of a partition. A disadvantage of convex partition is that overall parallelism is reduced. Without convexity constraint, we do not have to wait until all the inputs become available in order to start a partition. This implies that the latency incurred by synchronization of all inputs can be reduced and the overall parallelism thus increases at the cost of scheduling overhead (for context switch).

The convexity constraint can be strictly applied to the *fork-join* execution model. For example, in figure 3.7 and 3.8, if we consider each node in the graph as a partition, the corresponding sequence of forks and joins ensures that each node is run to completion once it is started. However, if any of the nodes (partitions) has its own set of partitions, in other words, if we allow hierarchical partitioning structure, the convexity constraint can no longer be satisfied. Recursive function calls also violate the the convexity constraint. If the *fork* operation contributes significantly to the overall execution time, it is also desirable for fork operations to be overlapped. Fork operations can be overlapped by hierarchical partitioning.

For example, in the flat partitioning shown in figure 3.9, *fork* operations are all done sequentially, and if we assume the execution time of each node (partition) and the fork operation are the same (one unit time as shown in the figure), the total execution time is 10. However in hierarchical partitioning, as in figure 3.10, fork operations are distributed over processors and the total execution time is thus reduced to 6.

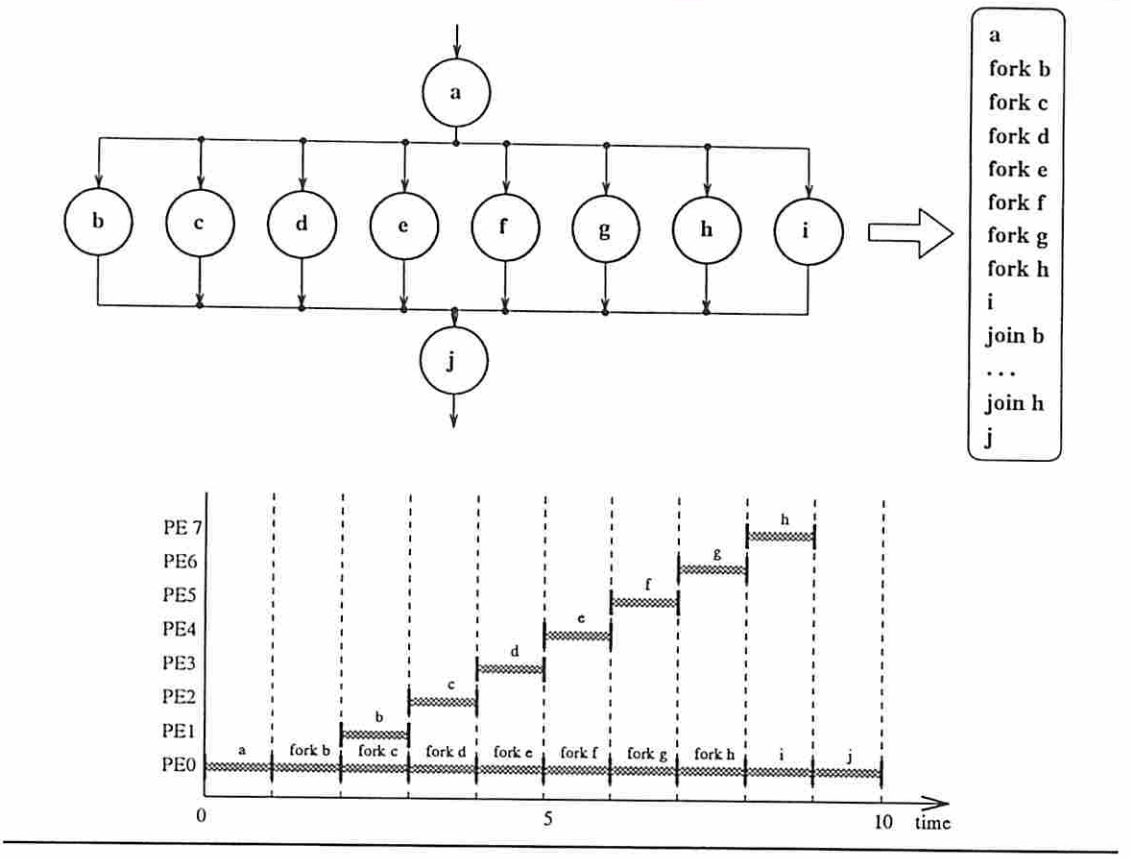


Figure 3.9: Flat partitioning in which fork operations are sequentialized. In the timing diagram, we assume the execution time of each node (partition) and the fork operations are all same (one unit time).

The hierarchical partitioning has another important advantage in our SISAL programming environment (section 2.2.1). The intermediate data-flow graph (IF1 [65]) which is produced by the SISAL front-end preserves the hierarchy of the program structure, *i.e.*, compound nodes of IF1 and their sub-graphs represent the hierarchical control structure of the SISAL program. Each of these compound

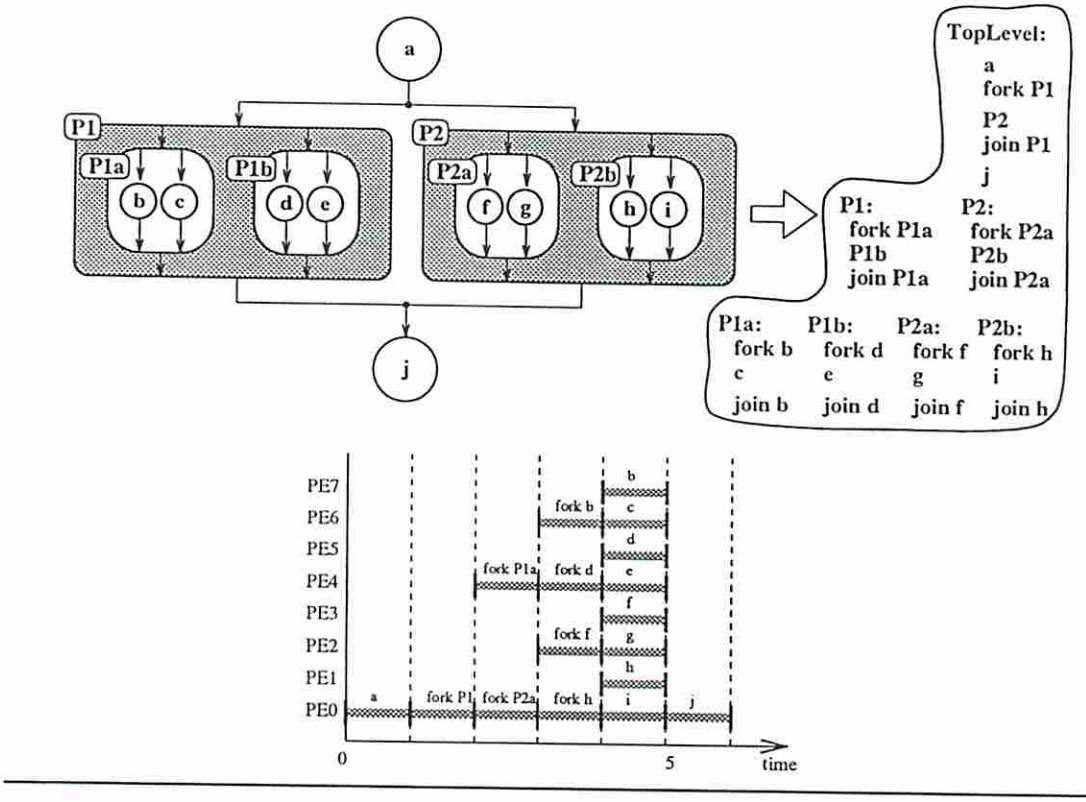


Figure 3.10: Hierarchical partitioning in which fork operations are also parallelized. In the timing diagram, we assume the execution time of each node (partition) and the fork operations are all the same (one unit time).

nodes (sub-graphs) can therefore be a good candidate for a partition in our coarse grain approach. If the number of IF1 simple nodes in a sub-graph is very large we may further decompose the sub-graph into several partitions. On the other hand, if the number of IF1 nodes in a sub-graph is relatively small, two or more sub-graphs can become a single partition for coarser granularity. However, as we have mentioned earlier, the right granularity must be determined by various system parameters.

3.3.4 Loop slicing

SISAL [55] (section 2.2.2) provides two types of loops, *iteration* (sequential loop) and *forall* (parallel loop). Although a scheme to extract runtime parallelism from sequential loops has been proposed [50], the implementation of the scheme in the aforementioned fork-join model incurs too much overhead due to the excessive occurrences of synchronizations at the fine grain level. In this research, we therefore investigate the parallelization of *forall* loop.

In the coarse grain approach, iterations of the loop-body are divided into several slices and the iteration within the slice is performed sequentially while slices are executed in parallel. The *number of slices* plays the most important role to reduce the execution time of the loop. In order for the loop slicing to be successful in improving the overall performance, the following condition should be satisfied as a bottom line:

$$\mathcal{S}_l(k, N) + \mathcal{E}_l(\lceil N/k \rceil) < \mathcal{E}_l(N)$$

where, $\mathcal{S}_l(k, N)$ is the runtime overhead to initiate k slices of the loop l with N iterations, and $\mathcal{E}_l(m)$ is the sequential execution time of m iterations.

In addition to the loop slicing, partitioning of *structure data* is a very crucial part for the performance of the parallel execution of a loop. For a distributed memory machine, the evaluation of the overhead function ($\mathcal{S}_l(k, N)$) must incorporate the communication time to copy partitioned structures over the interconnection network. Moreover, there are some cases in which the proper structure partitioning is not even possible due to the irregular access pattern. In this case,

the simplest solution is to copy the whole structure to all the processors. The better solution may be found with a centralized structure handling mechanism like *I-structure* [7] or the runtime support for *virtual shared memory* [43].

Chapter 4

Exploiting Dynamic Parallelism Using Parallel Function Calls

This chapter presents the extension to the Sisal compiler to enhance the applicability of OSC (section 2.2.3). This demonstrates that the *parallel function call* can be effective for the exploitation of coarse grain parallelism, particularly *dynamic parallelism* incurred by dyadic recursions. A general purpose shared memory multiprocessor system has been chosen ad the target architecture.

4.1 Divide-and-conquer problem – Parallelism by dyadic recursion

Many problems are solved by algorithms with recursive structures. In the execution of such algorithms, recursive calls are made one or more times to deal with closely related subproblems. These algorithms typically follow a *divide-and-conquer* approach. They break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of recursion [19] :

Divide : Divide the problem into a number of (typically two) subproblems.

Conquer : Conquer the subproblems by solving them recursively. If the subproblems are sufficiently small, however, just solve the subproblems in a straightforward manner.

Combine : Combine the solutions to the subproblems into the solution for the original problem.

Suppose we divide the problem into 2 problems, each of which is 1/2 the size of the original. The sequential execution time of a divide-and-conquer algorithm, $T_1(n)$, is defined by the following recurrence:

$$T_1(n) = \begin{cases} s & \text{if } n \leq m \\ 2T_1(n/2) + D(n) + C(n) & \text{otherwise} \end{cases}$$

where s is the time to solve the small problem (problem size $\leq m$) in a straightforward manner, $D(n)$ is the time to divide the problem into subproblems, and $C(n)$ is the time to combine the solutions of the subproblems.

The two subproblems are independent to each other. Their solutions are later combined to yield the solution to the original problem. Therefore, the two subproblems can be solved in parallel. Assuming that we have infinite number of processors and we take the function-call overhead into consideration, the theoretical execution time can be computed by the following recurrence:

$$T_\infty(n) = \begin{cases} s & \text{if } n \leq m \\ T_\infty(n/2) + D(n) + C(n) + 2F(n) & \text{otherwise} \end{cases}$$

where $F(n)$ is the time to make a parallel function-call. For simplicity, let $m = 1$ (it i.e. we divide the problem until the problem size is 1), and $D(n) = d, C(n) = c, F(n) = f$ for all n (constant regardless of the problem size), and also let $n = 2^k$. We then have the following recurrence:

$$T_\infty(k) = \begin{cases} s & \text{if } k = 0 \\ T_\infty(k-1) + d + c + 2f & \text{otherwise} \end{cases}$$

Therefore,

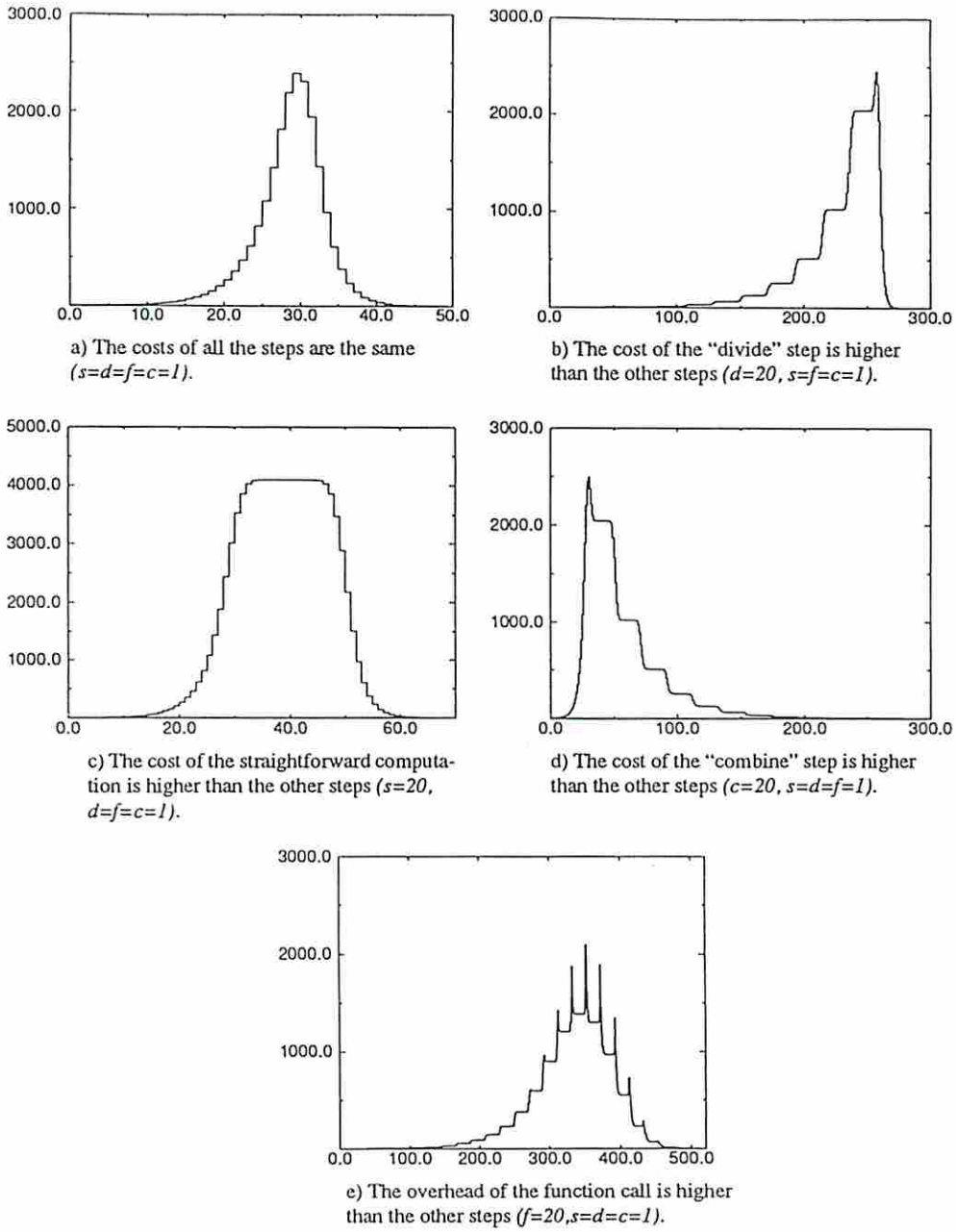
$$T_\infty(k) = (d + c + 2f)k + s$$

Based on this simplified model, we show various parallelism profiles of typical divide-and-conquer problem of size $n = 4096$ ($k = 12$) in figure 4.1. Figure 4.1a shows the parallelism profile of the generic divide-and-conquer problem, while the rest of the figures show the profiles when d,s,c and f are made to vary.

As can be seen in figure 4.1, the ideal case for parallel implementation is when each leaf node of the solution tree is relatively costlier than the other steps of the problem. The leaf node corresponds to the boundary condition for which we solve the problem in a straightforward manner rather than further subdividing the problem. Since the main objective of the divide-and-conquer solution is to make the subproblems simpler, as we go down the solution tree until we reach the minimal subproblem, the computation costs are correspondingly reduced. Thus, in the leaf node, we usually have very trivial computations, or, in some cases, no computations at all. In general, most divide-and-conquer problems will exhibit a parallelism behavior resembling that is shown in figure 4.1b or 4.1d. For example, *quick-sort* [51] takes more time in dividing the input set to two sub-sets (figure 4.1b), while *merge-sort* [51] spends more time in merging two sub-results (figure 4.1d). Aside from the actual computation costs, we must also consider some overheads for parallel invocation of the function. For some platforms, the overhead of resource allocation/deallocation for function call may overwhelm the *true* computation cost. This overhead is noticeable even for sequential machines, and can be a significant factor for parallel machines in which the resources are accessed by the computation units through some form of interconnection network. When the cost of function call dominates the overall execution time, the parallelism profile is less smooth and thus the usable parallelism is accordingly reduced (figure 4.1e).

4.2 A case study – Fast Fourier Transform (FFT)

In this section, I present the implementation of FFT based on the our programming environment (section 2.2.1), and also show the schemes to exploit the parallelism incurred by the dyadic recursion. The details are also presented in [71].



a) The costs of all the steps are the same *(s=d=f=c=1)*.

b) The cost of the "divide" step is higher than the other steps *(d=20, s=f=c=1)*.

c) The cost of the straightforward computation is higher than the other steps *(s=20, d=f=c=1)*.

d) The cost of the "combine" step is higher than the other steps *(c=20, s=d=f=1)*.

e) The overhead of the function call is higher than the other steps *(f=20,s=d=c=1)*.

Figure 4.1: Parallelism profiles of divide-and-conquer problems (X-axis and Y-axis denotes time and parallelism, respectively.).

4.2.1 Description of the Application

The *Discrete Fourier Transform* can be expressed as follows:

$$F_k = \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j$$

Let W be the following complex number:

$$W \equiv e^{2\pi i/N}$$

Then, F_k can be rewritten as follows:

$$\begin{aligned} F_k &= \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ik(2j)/N} f_{2j} + \sum_{j=0}^{N/2-1} e^{2\pi ik(2j+1)/N} f_{2j+1} \\ &= \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j} + W^k \sum_{j=0}^{N/2-1} e^{2\pi ikj/(N/2)} f_{2j+1} \\ &= F_k^e + W^k F_k^o \end{aligned}$$

F_k^e denotes the k^{th} element of the Fourier transform of the length $N/2$ formed from the even components of the original f_i's, while F_k^o is the corresponding transform of length $N/2$ formed from the odd components.

The recursive algorithm based on the above equation is shown in figure 4.2. [29, 60]

The time complexity of the above algorithm is $O(N \log N)$ with a single processor. However, parallelism can be exploited by the two recursive *fft* calls. Thus, in the ideal case, *i.e.* with a sufficient number of processors without any communication overhead, the time complexity reduces to $O(\log N)$.

4.2.2 Analysis of the Application using SISAL Tools

We code the FFT algorithm in SISAL and run it through DI (Debugger and Interpreter). DI is a debugging tool which enables us to do the followings:

Algorithm *fft*
Input:
f : array of N complex numbers, where N is a power of 2.
Output:
F : array of N complex numbers which is the Fourier transform of f.
Begin
1. Let f^e be the array of even components of f.
   Let f^o be the array of odd components of f.
2. Find $F^e = fft(f^e)$ and $F^o = fft(f^o)$.
3. Let W be $e^{2\pi i/N}$.
   For $k = 0, \ldots, N/2$, let $L_k = F_k^e + W^k F_k^o$.
   For $k = 0, \ldots, N/2$, let $U_k = F_k^e - W^k F_k^o$.
4. For $k = 0, \ldots, 2/N$, let $F_k = L_k$ and $F_{k+N/2} = U_k$. In other words, F is the concatenation of L and U.
End

Figure 4.2: The recursive FFT algorithm.

- To verify the functionality of the program. In other words, to check if each function produces the correct result.
- To observe the potential parallelism of the program.
- To obtain the simulated execution time on various number of processors.

The overall steps of the analysis using DI is shown in figure 4.3.

It is very intuitive that the algorithm itself must have sufficient parallelism to take advantage of its parallel execution. However, according to *Amdahl's law* [2], the parallel execution time of an algorithm is dominated by the parts where little concurrency can be exploited. Therefore, we must concern how useful the observed parallelism is. For example, no matter how good the peak parallelism is, when the sequential portion of the algorithm is dominating during the execution time, we cannot expect reasonable speedup with the parallel execution of the algorithm. As can be seen in figure 4.4a, our `fft()` program shows a reasonable shape of parallelism. The parallelism grows gradually to some point and reduces gradually toward the end of the execution. Thus, we can expect the program could make good utilization of a large number of processors. For example, if we assume the

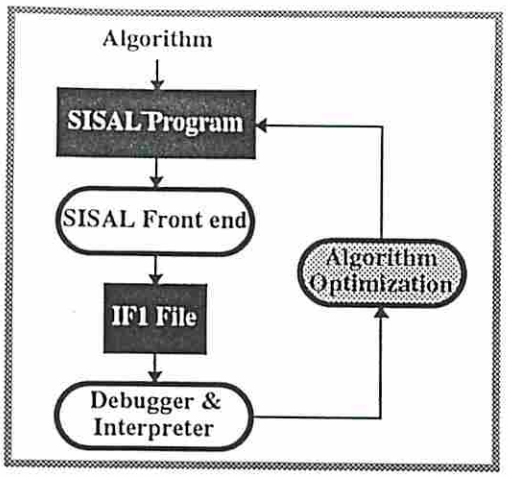


Figure 4.3: Steps of the experiment.

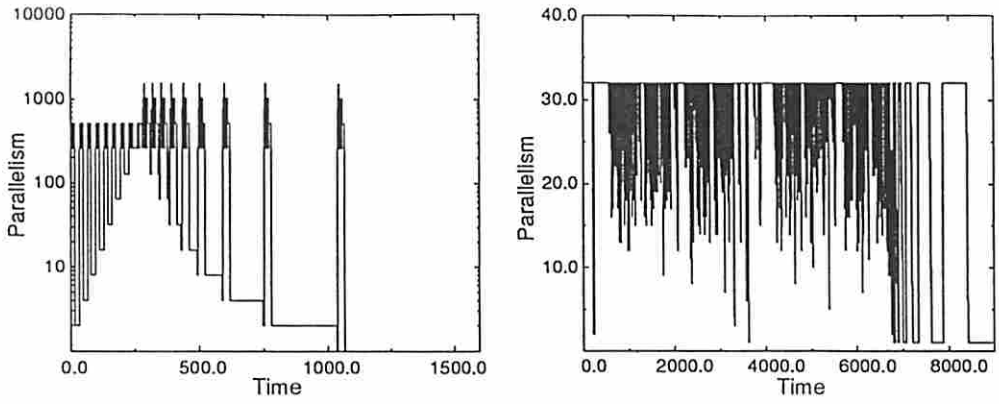
number of processors to 32, the histogram of clipped parallelism becomes near rectangle-shape as shown in figure 4.4b.

We ran the FFT for various data size from 16 points to 2048 points on several H/W configurations whose number of processors varying from 1 to 512. We also presented the total number of operations[1] (table 4.1), execution time (table 4.2), and finally the speedups (table 4.3). The columns in the tables corresponds to the data size (number of sampled points) and the rows corresponds to the number of processors. The comparison of ideal speedups between different data-sets is plotted in figure 4.5.

FFT points	16	64	128	256	512	1024	2048
Operations	1746	10098	23282	52772	117746	260082	569330

Table 4.1: Total number of operations

[1] the total number of IF1 nodes executed in our simulation



a) Potential parallelism of the 512pts FFT with an infinite number of PEs.

b) Potential parallelism of the 512pts FFT with 32 PEs.

Figure 4.4: Potential parallelism of FFT

	16	64	128	256	512	1024	2048
1	3421	19933	46045	104413	233437	516061	1130461
4	975	5392	12272	27530	61078	134244	292742
8	577	2879	6429	14279	31556	69071	150164
16	340	1612	3478	7618	16613	36119	78092
32	272	994	1995	4210	8942	19310	41373
64	271	588	1209	2456	5147	10867	23013
128	271	495	796	1596	3160	6585	13739
256	271	493	688	1135	2243	4425	9029
512	271	493	684	1006	1743	3429	6742

Table 4.2: Simulated execution time of FFT

	16	64	128	256	512	1024	2048
4	3.5	3.7	3.8	3.8	3.8	3.8	3.9
8	5.9	6.9	7.2	7.3	7.4	7.5	7.5
16	10.1	12.4	13.2	13.7	14.1	14.3	14.5
32	12.6	20.1	23.1	24.8	26.1	26.7	27.3
64	12.6	33.9	38.1	42.5	45.4	47.5	49.1
128	12.6	40.3	57.8	65.4	73.9	78.4	82.3
256	12.6	40.4	66.9	92.0	104.1	116.6	125.2
512	12.6	40.4	67.3	103.8	133.9	150.5	167.7

Table 4.3: Speedup of FFT for various data size

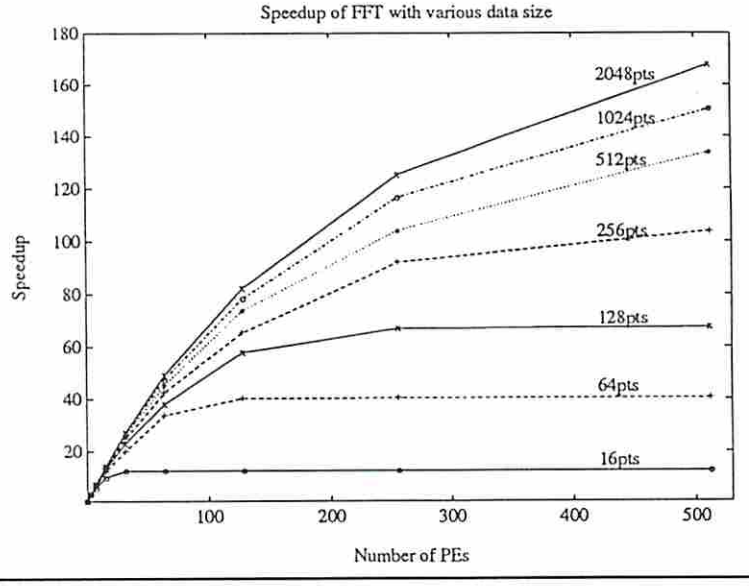


Figure 4.5: Comparison of ideal speedups.

4.2.3 Recursive FFT on the Sequent Balance using Parallel Function Call

The target machine on which we chose to run our SISAL implementation of the recursive FFT is the Sequent Balance shared-memory multiprocessor. This multiprocessor consists of 16 ns32032 processors which are connected by a common bus and share a single pool of memory. All processors (identical), memory modules, and I/O controllers are plugged into a single high-speed bus, and thus the whole system can be easily expanded. The multiprocessor is run under the DYNIX operating system, a multiprocessor version of UNIX. Therefore, unlike many other multiprocessor systems in which the main computation unit operates as a back end processing unit, each processor can cooperate to run not only the user code but also the kernel code in parallel. [59] For our implementation we also use *gang scheduling* which enables the user to claim all the processors before a program can commence execution. Release of the processors takes place only after the application has completed. This special scheduling scheme is particularly useful to measure performance.

For our SISAL implementation, we use OSC (Optimizing SISAL Compiler) to generate a portable C-code (See Section 2.2.1 for details about OSC). OSC mainly targets shared-memory systems, and it performs a certain level of parallelization. However, for our application, where the potential parallelism is mainly obtained by the dyadic recursion, the parallelization of OSC does not give us a reasonable speedup, since OSC performs loop slicing for parallel execution of the user program. Therefore, we introduce a *parallel function call* mechanism to the original OSC runtime system.

The following two primitives are used to invoke a function in parallel:

1. **ParallelCall** : Build a new task for a function invocation and returns a pointer to a new task.
2. **ParallelJoin** : Wait until the invoked function is completed.

In the example of a parallel invocation of two functions shown in figure 4.6, functions `f1` and `f2` are called in parallel. The caller continues its execution until it reaches `ParallelJoin()`. The `ParallelJoin()` should be called before the

```
...
t1 = ParallelCall (f1, arg1, arg2, ...);
t2 = ParallelCall (f2, arg1, arg2, ...);
...
ParallelJoin(t1);
ParallelJoin(t2);
```

Figure 4.6: Example of Parallel function call.

result of the corresponding function call is needed. The caller itself, while waiting for the called function to be completed, attempts to execute another task if any.

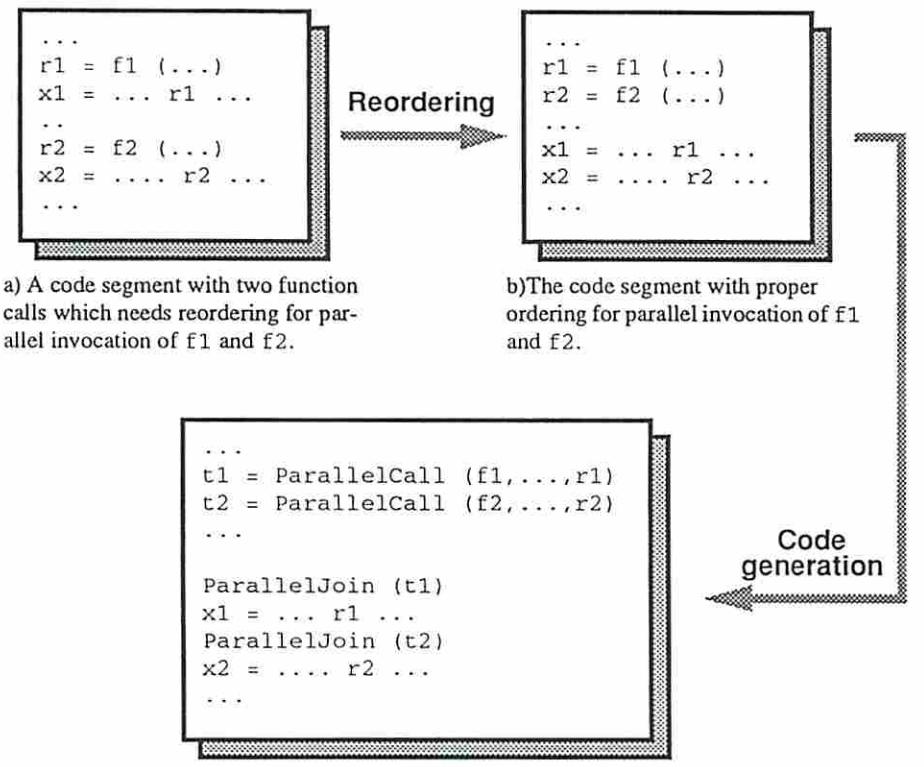
We still use OSC to generate C-code from a SISAL program. After generating C-code, we have to reorder the program statements to exploit the parallel function calls. Consider the code segment shown in figure 4.7a. If functions `f1` and `f2` are independent and can be called in parallel, the code should be reordered as shown in figure 4.7b. Further, using `ParallelCall` and `ParallelJoin`, the final code would be as shown in figure 4.7c.

As can be seen in the above example, to fully exploit the function call parallelism, we must provide a mechanism to reorder the statements of the target language. However in a real implementation, the reordering should be done during the partitioning step. The code generator will produce the correct parallel code according to the partitioned data-flow graph. In the example above, if the partitioning is done properly, the statements calling `f1` and `f2` should belong to different partitions. The code generator can therefore produce the correct code to execute these two partitions in parallel.

The execution time of the recursive FFT is shown in Table 4.4. In this table, we also showed the result obtained by the parallelization of the original OSC, *i.e.*, loop slicing. As shown in the table, for our recursive FFT, the parallel function call is much more effective than loop slicing. The corresponding speedup is also shown in Table 4.5 and in Figure 4.8.

We also have run the same program on the following platforms:

1. HP : HP9000-425t with 25MHz 68040 and a proprietary floating-point co-processor running HPUX.



a) A code segment with two function calls which needs reordering for parallel invocation of `f1` and `f2`.

b)The code segment with proper ordering for parallel invocation of `f1` and `f2`.

c) The final target code using `ParallelCall` and `ParallelJoin`.

Figure 4.7: An example of code conversion for parallel invocations of two functions.

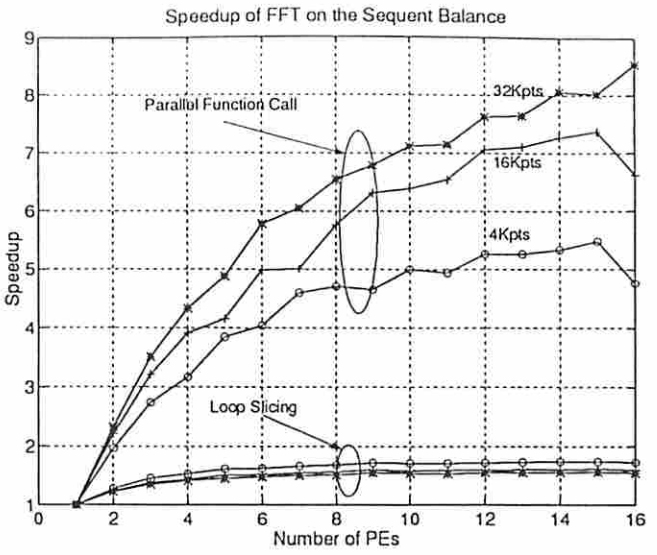


Figure 4.8: Speedup of the recursive FFT on the Sequent Balance

	Loop Slicing			Parallel Function Call		
#PE	4096pts	16384pts	32768pts	4096pts	16384pts	32768pts
1	42.2	224.4	521.1	40.0	214.0	501.3
2	33.0	181.7	426.3	20.4	97.0	215.6
3	29.0	162.3	387.1	14.6	66.5	142.6
4	27.6	156.5	369.5	12.6	54.7	115.8
5	26.2	149.0	359.8	10.4	51.5	102.7
6	26.0	148.2	353.3	9.9	42.9	86.7
7	25.4	145.0	347.4	8.7	42.7	82.8
8	25.1	143.7	346.4	8.5	37.1	76.6
9	24.6	140.2	337.0	8.6	33.9	73.9
10	24.7	142.4	339.8	8.0	33.5	70.4
11	24.7	141.0	341.3	8.1	32.7	70.1
12	24.6	140.7	336.8	7.6	30.3	65.7
13	24.4	140.1	335.8	7.6	30.1	65.5
14	24.3	140.2	335.1	7.5	29.4	62.2
15	24.3	138.7	334.6	7.3	29.0	62.5
16	24.4	140.9	337.9	8.4	32.3	58.7

Table 4.4: Execution time of recursive FFT on the Sequent Balance

	Loop Slicing			Parallel Function Call		
#PE	4096pts	16384pts	32768pts	4096pts	16384pts	32768pts
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.3	1.2	1.2	2.0	2.2	2.3
3	1.5	1.4	1.3	2.7	3.2	3.5
4	1.6	1.4	1.4	3.2	3.9	4.3
5	1.6	1.5	1.4	3.8	4.2	4.9
6	1.7	1.5	1.4	4.0	5.0	5.8
7	1.7	1.5	1.5	4.6	5.0	6.0
8	1.7	1.6	1.5	4.7	5.8	6.5
9	1.8	1.6	1.5	4.7	6.3	6.8
10	1.7	1.6	1.5	5.0	6.4	7.1
11	1.8	1.6	1.5	4.9	6.5	7.2
12	1.8	1.6	1.5	5.3	7.1	7.6
13	1.8	1.6	1.6	5.3	7.1	7.7
14	1.8	1.6	1.6	5.3	7.3	8.1
15	1.8	1.6	1.6	5.5	7.4	8.0
16	1.8	1.6	1.5	4.8	6.6	8.5

Table 4.5: Speedup of recursive FFT on the Sequent Balance

2. SUN4 : Sun Sparc system 4000 running SUN OS.
3. SUN3 : Sun3/100 with M68881 floating point processor.

The execution time is shown in Table 4.6.

#Pts	Balance(1PE)	HP	SUN4	SUN3
4096	42.2	1.9	0.9	19.5
16384	224.4	10.3	4.6	92.4
32768	521.1	17.9	7.4	185.0

Table 4.6: Execution time of FFT on the various sequential machines

To compare the general performance of each machine, we also presented the execution results of *Whetstone* and *Dhrystone* in Table 4.7 and Table 4.8. The *Whetstone* benchmark is used to test the floating-point performance while the *Dhrystone* is used to test mainly the performance of general system calls.

Balance(1PE)	HP	SUN4	SUN3
37.7	2.1	0.9	23.9

Table 4.7: Time to perform 10 Million Whetstone instructions.

Balance(1PE)	HP	SUN4	SUN3
1098	25000	33333	3448

Table 4.8: Number of Dhrystone instructions executed per second

4.2.4 Analysis of the results

As shown in the results, the parallel function call approach is an effective way of exploiting parallelism incurred by recursion. However, some remarks must be made about the performance results. First of all, there is an order of magnitude difference between the speedup we obtained on the Sequent Balance and the ideal

speedup observed by running the IF1 graph using DI (figure 4.5 and figure 4.8). The major difference between the two is the granularity. In IF1, we exploit the fine-grain parallelism at the IF1 instruction level, while we applied function-level parallelism in our implementation on the Sequent Balance. Since the simulation with DI assumes no extra costs in handling resources and performing synchronization between IF1 nodes, we are guaranteed that the finer the grain, the higher the speedup.

Secondly, although the recursive FFT has high function-level parallelism which grows exponentially as we go down further to the subproblems, the cumulative parallelism is not as high as the peak parallelism. For example, in our simulation of recursive FFT based on the parallel function call, the peak parallelism is over 2000 (with 4096 points), but as can be seen in figure 4.9, for almost half of the total execution time, the parallelism remains at one. Therefore, as shown in figure 4.10, the speedup of the FFT reaches very quickly a certain limit which is rather low compared to the number of PEs, due to Amdahl's law [2]. Simulated execution time and the speedup based on the parallel function call scheme are also shown in table 4.9 and table 4.10, respectively.

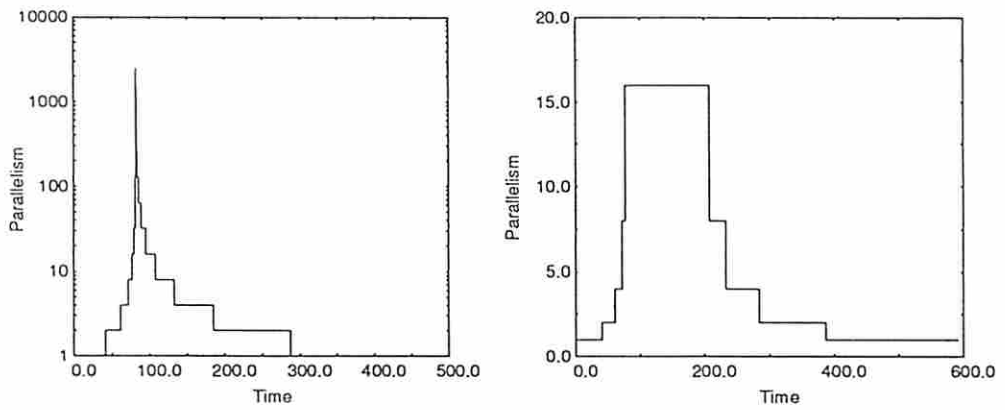


Figure 4.9: The parallelism profile of FFT based on function level parallelism. The FFT has been performed on 4096 points and the left graph shows maximum potential parallelism while the right one is the clipped version at 16 PEs.

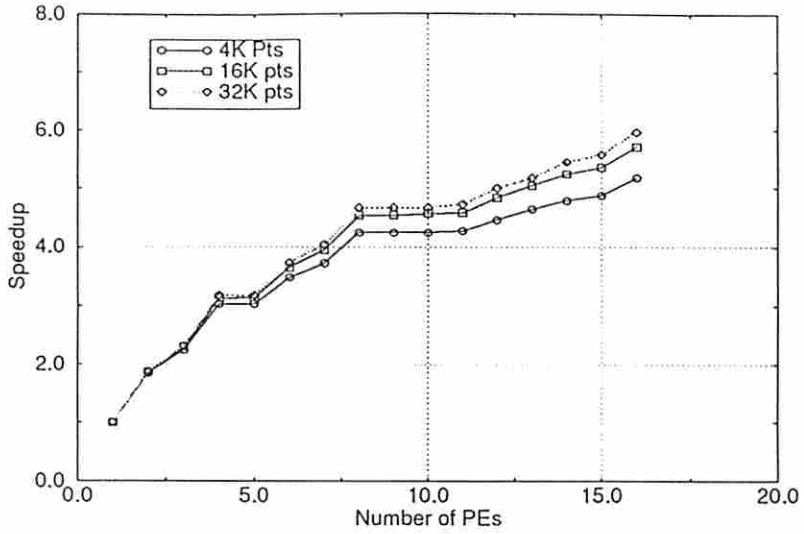


Figure 4.10: Speedup of the recursive FFT (4096 points) based on parallel function call — Simulated results using CSIM package.

#PE	4096pts	16384pts	32768pts
1	3071.99	14254.07	30474.23
2	1658.90	7618.58	16220.18
3	1367.09	6205.50	13148.21
4	1013.82	4546.62	9584.70
5	1013.85	4531.87	9584.74
6	883.30	3901.54	8171.63
7	825.70	3609.70	7526.52
8	722.02	3133.54	6512.74
9	722.02	3133.54	6512.74
10	722.02	3112.26	6512.74
11	717.22	3103.81	6428.86
12	686.61	2939.64	6077.65
13	660.20	2818.26	5867.70
14	639.54	2711.23	5575.86
15	628.96	2653.60	5445.28
16	591.50	2488.46	5099.66

Table 4.9: Simulated execution time of the recursive FFT.

#PE	4096pts	16384pts	32768pts
1	1.00	1.00	1.00
2	1.85	1.87	1.88
3	2.25	2.30	2.32
4	3.03	3.14	3.18
5	3.03	3.15	3.18
6	3.48	3.65	3.73
7	3.72	3.95	4.05
8	4.25	4.55	4.68
9	4.25	4.55	4.68
10	4.25	4.58	4.68
11	4.28	4.59	4.74
12	4.47	4.85	5.01
13	4.65	5.06	5.19
14	4.80	5.26	5.47
15	4.88	5.37	5.60
16	5.19	5.73	5.98

Table 4.10: Simulated speedup of the recursive FFT.

The histogram shown in figure 4.9 is very close to figure 4.1d in which the *combine* operation is the dominant part of the computation. Consider the combine part of our recursive FFT algorithm shown in figure 4.11.

---

3 Let W be $e^{2\pi i/N}$.
For $k = 0, \ldots, N/2$, let $L_k = F_k^e + W^k F_k^o$.
For $k = 0, \ldots, N/2$, let $U_k = F_k^e - W^k F_k^o$.
4 For $k = 0, \ldots, 2/N$, let $F_k = L_k$ and $F_{k+N/2} = U_k$. In other words, F is the concatenation of L and U.

Figure 4.11: The combine part of the recursive FFT algorithm.

The two steps in figure 4.11 are actually merged into a single *forall* loop in the SISAL implementation. The loop size, N_l, at the level l in the recursion tree ($l = 0$ at the root level) is defined by:

$$N_l = \frac{N_p}{2^l}$$

where N_p is the problem size, in our case, the number of sampled points for FFT. As can be easily understood by the above equation, the upper level of the tree comprises larger loops. In other words, for fewer instances of functions, there will be larger *forall* loops. In our scheme, we only parallelize function call, and therefore, in the upper level where the number of outstanding function calls is very small, we have many PEs idling. This is the main reason why the potential parallelism remains low for the most of the time during the execution of FFT (figure 4.9).

Chapter 5

SIMD *vs.* MIMD : From the Perspective of Sisal Implementors

This chapter addresses the issues of compilation and execution of a functional program, SISAL (Streams and Iterations in a Single Assignment Language), on the MP-1^{TM} SIMD (Single Instruction-stream Multiple Data-stream) parallel machine. SISAL has been successful on many shared memory multiprocessors (SMM) as well as sequential machines. However, the compiler has not been available for distributed memory multiprocessors (MM) and SIMD machines. The original goal of the project presented in this chapter is to assess the programmability of signal processing applications on both SMM and SIMD machines. The application has been coded in SISAL and executed on both type of machines. In order to implement the SISAL code on the MP-1 SIMD machine, we proposed the compilation schemes from SISAL code to MP-1's data-parallel C language (MPL). The experimental results proved that for a certain type of applications (such as signal processing applications presented in this chapter), a functional programming paradigm is indeed very effective on SIMD machines in terms of both programmability and performance. We also contrast various aspects of programming MIMD and SIMD machines as our conclusion.

5.1 Introduction

The major goal of this work is to assess the programmability and the performance of signal processing applications on a SIMD machine, specifically, in the functional

programming environment. In a pure functional programming language [8], parallelism need not be expressed by the programmer since it is implicit. Because a program written in a pure functional language is side-effect-free, the relative order of the program statements is not as significant as in an imperative language. Each statement can be executed in any order as long as the data dependencies are respected. Therefore, it is easier for the compiler to extract parallelism out of a program written in a functional programming language. Moreover, programs can be easily ported to other platforms since details about specific machines or operating systems can be hidden from the programmer.

We chose a functional language, SISAL (Streams and Iterations in a Single Assignment Language) [55], for the parallel implementation of our application. Besides the SISAL is a functional language it has already been available on many shared memory multiprocessor systems as well as many sequential machines. However, there is no SISAL compiler available for any SIMD machine. Our research is therefore mainly focused on how a SISAL program can be effectively implemented on a data-parallel machine such as MP-1/MP-2.

In this report, we present many issues of programming a signal processing application in the SISAL programming environment. In the rest of this report, we will present the features of SISAL programming environment and the parallel programming on MP-1 system. We will also describe our target application and will show how it can be implemented on MP-1 coupled with SISAL programming environment. In the last part of this report, the comparison of programming on two different types of parallel machines, MIMD *vs.* SIMD, will be presented.

5.2 Data-parallel Programming on MP-1

In this section, we describe the overall programming environment of the MP-1 system from both the hardware and the software perspectives [53, 54].

5.2.1 Overview of the MP-1 System

The MasPar MP-1 massively parallel computer consists of a front-end host workstation and a back-end *data-parallel unit (DPU)*. The DPU has two components:

an Array Control Unit (ACU) and an array of PEs (Figure 5.1). Each component can be described as follows:

- *Front End* : The front end mainly handles the user interface, inputs/outputs, and the normal functions which a conventional workstation would provide, such as compiling and network functions. The initial data setup and the final collection of results can also be performed by the front end. Incidentally, it should be noted that the front end of the MP-1 is a DECstation running the Ultrix operating system.

- *DPU* : The DPU handles most of the computations. It can be viewed as a massively parallel machine (PE Array) plus an Added Scalar Processor (ACU).

  - *ACU* : The main job of the ACU is to decode and broadcast instructions to all the PEs in the PE array. The ACU also has a RISC processor that can operate on scalar variables.

  - *PE Array* : The Processor Element (PE) Array forms the computational core of the MP-1 system and includes up to 16,384 PEs operating in parallel. Each PE is a custom-designed RISC processor with 64KB dedicated data-memory and 40 32-bit registers. The data in the PE memory space is distributed, while each instruction from the ACU is executed on all the PEs simultaneously.

The highly optimized communication between neighboring PEs is achieved by the *Xnet* while a flexible global communication is performed by the *router*. The communication primitives will be later described in section 5.2.2.3.

5.2.2 Programming the MP-1 System

The MP-1 massively parallel computer provides the programming tools to support data-parallel programming model. These include the MasPar Programming Language (MPL) as well as its own runtime libraries for communication. In this section, we describe the concept of data-parallel programming and the programming tools of the MP-1 system.

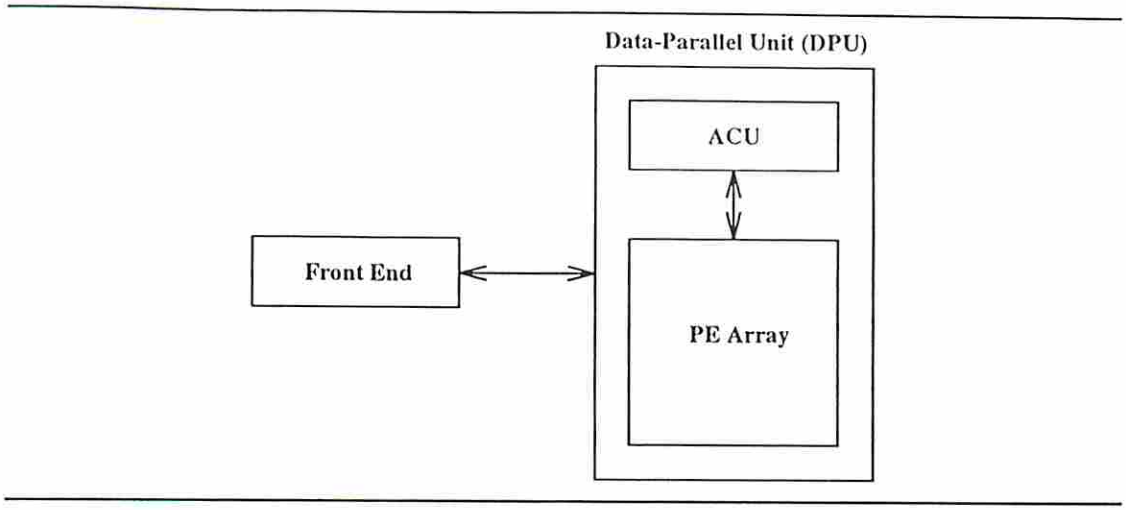


Figure 5.1: Overall structure of the MP-1 system

5.2.2.1 Data-parallel programming

Generally speaking, there are two basic programming models which programmers can employ to take advantage of parallel systems:

1. Data-parallel
2. Control-parallel

In the *data-parallel* model, there is a large data set that needs to be processed, and each processor executes the same set of instructions on the different data in the set (SIMD: Single Instruction Stream Multiple Data Streams). The main feature of the architecture for data-parallel model is that it includes a large number of rather simple processors. In order for a program to be executed efficiently in this model, the synchronization between processors should occur in a very regular pattern. Thus, by highly optimizing a certain type of communication pattern (*e.g* between neighboring processors), the overall communication overhead can be well-matched to the speed of the processor.

In the *control-parallel* model, each processor executes separate processes/functions to solve either independent problems or cooperate on the same problem (MIMD: Multiple Instruction Stream Multiple Data Stream). This model is more flexible than the data-parallel model, and its applicability is wider than the data-parallel model. However, the underlying architecture for the model usually results in more

complex control and communication structures. Moreover, due to the complexity, it is not yet feasible to employ very large numbers of processors in a single system as easily as in data-parallel architectures.

When using either the data-parallel or the control-parallel model, the algorithms need to be redesigned to take advantage of the corresponding model. In the case of data-parallel programming, the algorithms should be designed for large amounts of data, and it assumes that each data element is assigned to one processor. Therefore, as we increase the data size, the usable parallelism increases accordingly and the program can be scaled up easily to provide increased performance. For example, consider the following expression:

For $i = 0 \ldots N-1$

$$C[i] = f(A[i], B[i])$$

where f is an arbitrary function of two inputs which we need to evaluate. In the data-parallel programming model, the above expression can be simply implemented as follows:

$$c = f(a, b)$$

assuming the elements of A and B are distributed among the PEs such that the value of a on PE_i is indeed the value of $A[i]$ (Figure 5.2).

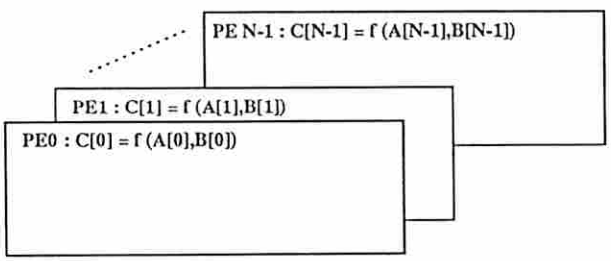


Figure 5.2: The function f is evaluated in parallel for each element of A and B across the PEs

5.2.2.2 Overview of MPL

The MasPar Programming Language (MPL) is used to program the DPU. MPL is based on ANSI C with the added statements, keywords, and library functions to support data parallel programming.

The key features of MPL are its support of the following data types:

- The *Plural* data type is used to specify data storages in DPU, *i.e.* parallel data, while without *plural* the storage is defined in the ACU.
- The *Plural* expressions can be defined by any arithmetic and addressing operations on the *plural* data type. For example, if k, i, j are plural types, the expression, `k = i + j`, is executed on all the active PEs in the DPU.
- The semantics of the SIMD control statement are implemented by the concept of the active set. The active set is the set of PEs that is enabled at any given time during the execution. This set is defined by conditional tests in users program. For example, consider the code segment shown in Figure 5.3.

```
plural int i,j,k;
...
if ( i ¿ j)
    k = i - j;
else
    k = j - i;
...
```

Figure 5.3: An example of activeness control in MPL.

When the first statement is executed, the active PEs are those in which the condition, `i > j`, is true, and for the second statement the active set becomes complemented. At program startup, all PEs are active and the active set varies depending on the program's control structure.

Consider the following example once again:

For $i = 0 \ldots N - 1$

$$C[i] = f(A[i], B[i])$$

Now, assuming the number of elements is less than the number of PEs, we can write an MPL code as shown in Figure 5.4:

```
1. float A[N],B[N],C[N];
2. plural float a,b,c;
   ...
3. if (iproc ¡ N) {
4.     a = A[iproc], b = B[iproc];
5.     c = f (a,b);
6. }
7. for (i=0;i¡N;i++)
8.     C[i] = proc[i].c;
   ...
9. plural float f (plural float a,plural float b)
   ...
```

Figure 5.4: An example of an MPL code for parallel evaluation of function f.

In step 2, we define a, b, c as a parallel data whose storage is defined in all the PEs. Steps 4 and 5 are executed on those PEs whose number is less than N. Note that the function f is declared also as a parallel function in order for each PE to be able to call the same function simultaneously (step 9). After the parallel evaluation of f is done, the data is collected by C in steps 7 and 8. If there are any further computations on the result of f, the data collection will be deferred until all the parallel computations have completed on all PEs.

5.2.2.3 Communication

MPL provides a direct control over the PE-to-PE communication. There are two kinds of communication mechanisms available: near-neighbor regular communication via the *Xnet* communication network, and random communication via the global *router*.

Xnet

Xnet is faster than the global router. It is therefore advantageous to use Xnet for inter-PE communication whenever possible. Xnet connects each PE to

its neighboring PEs in a 2-D mesh. The overall topology of Xnet is the 8-way toroidal wrap-round, *i.e.*, each PE is connected to 8 neighboring PEs (Figure 5.5). The *Xnet* library function is used to implement Xnet communication in MPL in the following form:

xnetDD[*distance*].*variable*

The above function designate the *variable* in the PE which is *distance* away in the *DD* direction. *DD* can be one of *N, NE, E, SE, S, SW, W* and *NW*, each corresponding to one of the eight directions.

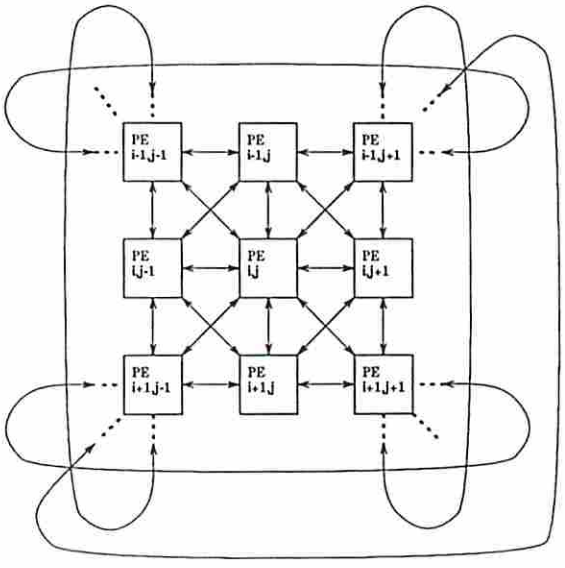


Figure 5.5: 8-way toroidal wrap around topology of Xnet.

As an example of using Xnet, we chose an image processing algorithm in which one is to average pixel values with 8 neighbors for each pixel[1]. This can be achieved with the MPL code shown in figure 5.6.

Global Router

The *global router* allows PEs to directly access any other PE in the PE array. It is mainly used for the communication patterns that are not regular, especially

[1]The pixel averaging is one of the popular and the simplest ways for anti-aliasing in computer image generation [18].

```
...
1. pixel = pixel + xnetW[1].pixel + xnetE[1].pixel;
2. pixel = pixel + xnetN[1].pixel + xnetS[1].pixel;
3. pixel = pixel/9;
...
```

Figure 5.6: An example of Xnet: Averaging pixels with 8 neighbors.

when the communication pattern is computed at run time. The format of the *router statement* is as follows:

router[*PEnum*].*variable*

The above function designates the *variable* in the PE *PEnum*.

There is one router channel for each set of 16 PEs. Hence there is a possibility of contention as the router channel works on a *first-come-first-served* basis. Unlike Xnet, the communication time of the router is not dependent on the distance of the two communicating PEs. In general, Xnet performs better for close neighbors (whose distance is less than 32) than the router. However, the algorithms requiring irregular or complex communication patterns will be solved better by using the router.

5.3 SISAL Programming on MP-1

As mentioned earlier in this report, there is no SISAL compiler available for any data-parallel machine. We therefore designed schemes to translate SISAL programs into MPL (a data-parallel version of C). The steps of translating a SISAL program into the corresponding MPL code can be described as follows:

1. Compile the SISAL program into an IF1 graph — We use the front-end of OSC (Optimizing SISAL Compiler) to generate the IF1 graph.

2. Perform traditional scalar optimization and structure optimizations. — The optimizer of OSC can be used for this purpose. It performs data-dependent analysis and also optimizes the structure (array) handling operations.

3. Transform the IF1 graph for the exploitation of data-parallelism. — In this step, parallel *forall* loops are transformed for data-parallel execution. This part is indeed the major contribution of our work. We propose a transformation scheme which will be discussed later in this report.

4. Generate MPL code. — Finally, we generate MPL code from the transformed data-flow graph.

The overview of the translation is also show in figure 5.7. In the figure, the shaded steps are already implemented in OSC.

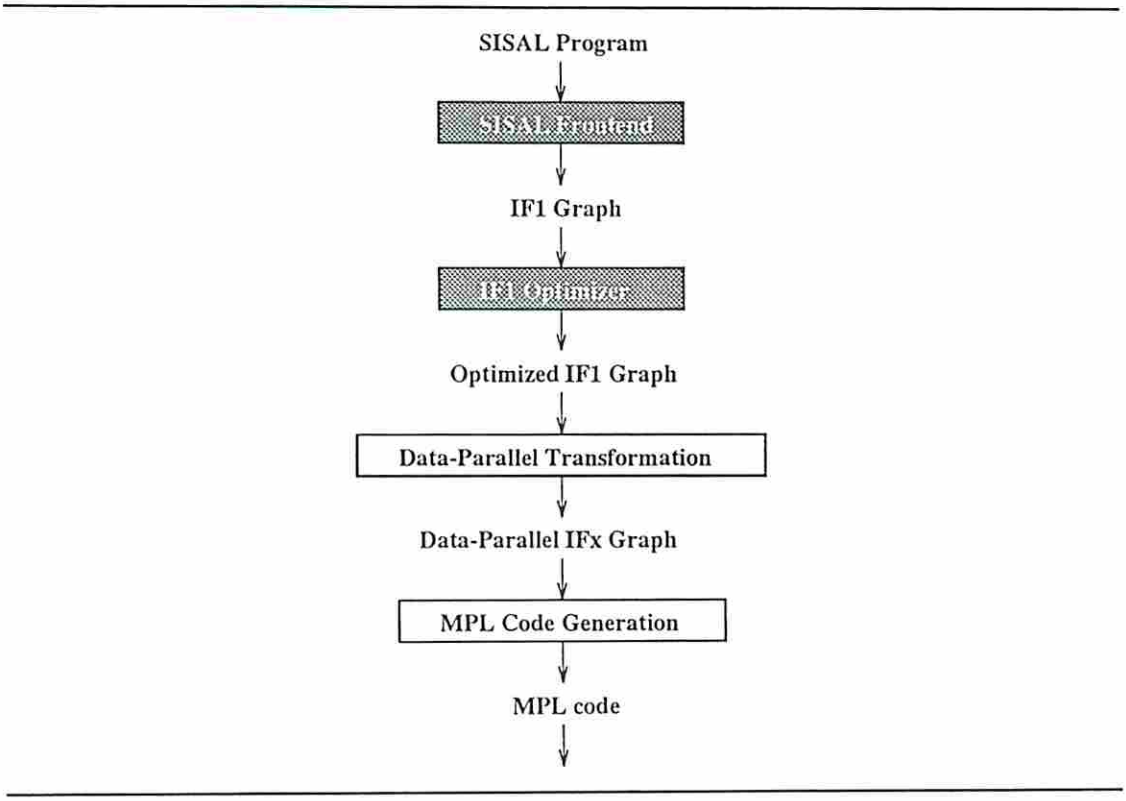


Figure 5.7: The overview of the translation steps.

5.3.1 Data-Parallel Transformation of IF1 Graph

In an IF1 graph, the main construct in which we can exploit data-parallelism is the *forall* loop. Moreover, on data-parallel machines like MP-1/MP-2, a large data

array must be allocated across the PE array due to the limitation of memory space on the Array Control Unit (ACU). Therefore, even the array is to be implemented for the sequential execution only, it is reasonable to assume that all the arrays are simply allocated on all PEs across the PE array. In describing our transformation scheme, we also assume that all the input arrays are already allocated over the PEs based on *cut-and-stack* mapping [53].

A Forall loop consists of three subgraphs, the *generate* graph, the *body* graph and the *return* graph. For example, the SISAL program shown in figure 5.8 is translated into the IF1 graph shown in figure 5.9.

```
function foo (v:OneDim;n:integer returns OneDim)
     for i in 0,n-1
          r := f (v[g(i)],v[h(i)]);
     returns
          array of r
     end for
end function
```

Figure 5.8: A simple SISAL program with forall loop.

Transforming *MemAlloc* and *AGatherAT*

MemAlloc is a special node of IF1 (IF2) used to specify the allocation of a memory block. In a shared memory multiprocessor system, MemAlloc simply allocates a contiguous block of memory for an array. However, in the data-parallel programming paradigm, when an array must be allocated across the PE array, only a small portion of the array can be assigned to each PE. For this purpose, we thus introduce a new construct called *pMemAlloc* (figure 5.10). The input to *pMemAlloc* is the total size of the array and the output is the pointer to the newly allocated slot on each PE. The size of the space allocated on each PE is also determined by the total number of processors ("nproc").

The *AGatherAT* node is used to collect the result of the loop-body. We simply replace *AGatherAT* with *pAGatherAT* in accordance with the output of *pMemAlloc*.

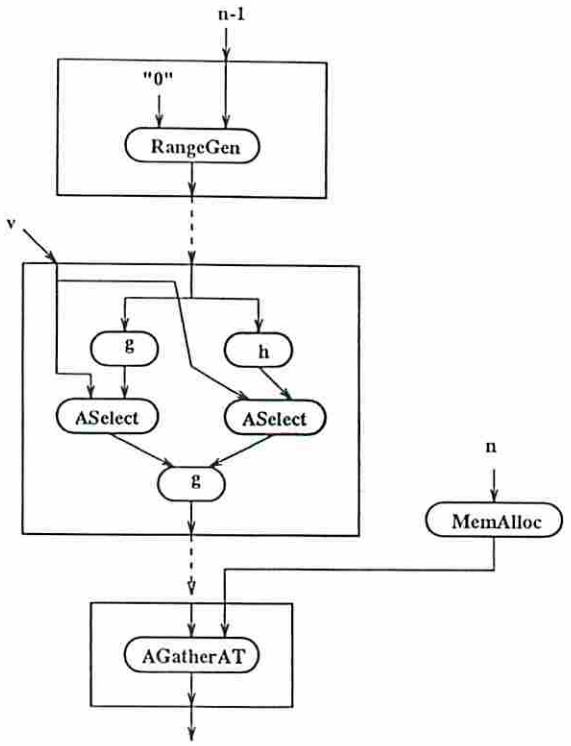


Figure 5.9: A simple IF1 graph with forall loop.

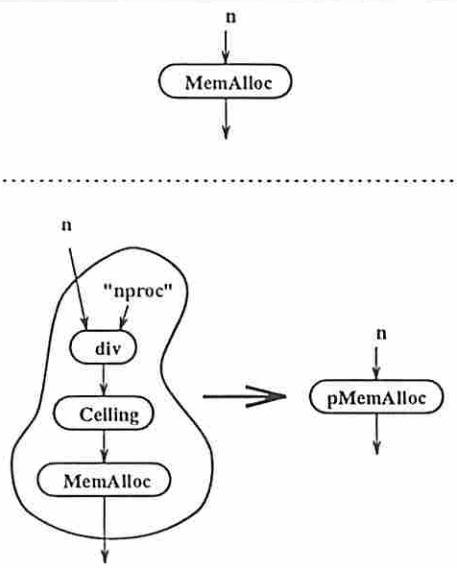


Figure 5.10: Transforming *MemAlloc* for array distribution across the PE array.

Transforming *RangeGenerate*

RangeGenerate initiates the parallel invocation of a loop-body by generating iteration indices. Just as in the case of *MemAlloc*, on each PE, only a small number of iterations are performed. This number is based on the total number of iterations and the number of available PEs (nproc). The *RangeGenerate* node is replaced with two new constructs called *SlashBounds* and *pRangeGenerate* as shown in figure 5.11.

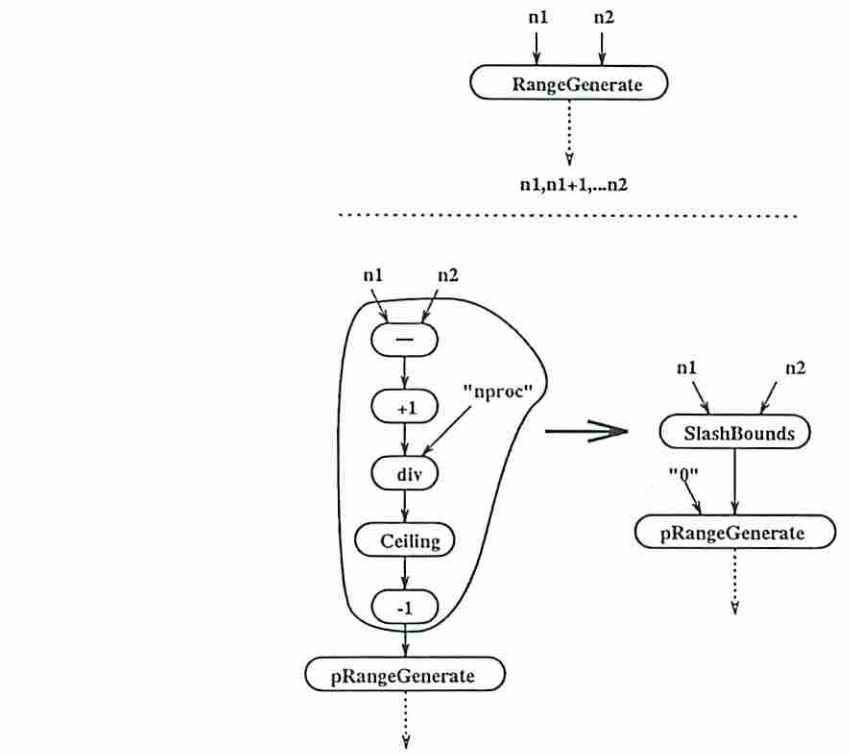


Figure 5.11: Transforming *RangeGenerate* for a data-parallel forall loop.

In most cases, the array index is used in the body of a forall loop. It is therefore necessary to retrieve the original index out of the local index (of the sliced array on each PE). Retrieval of the index can be efficiently performed using a bit-wise shift operation only if the number of processors is a power of two. For example, if the number of processors is 2^k, and the local index is i_p, the original index i can be obtained by shifting i_p k bits to the left and adding the current PE number. This

transformation is shown in figure 5.12, where "nshift" and "iproc" corresponds to k and *PE number* respectively.

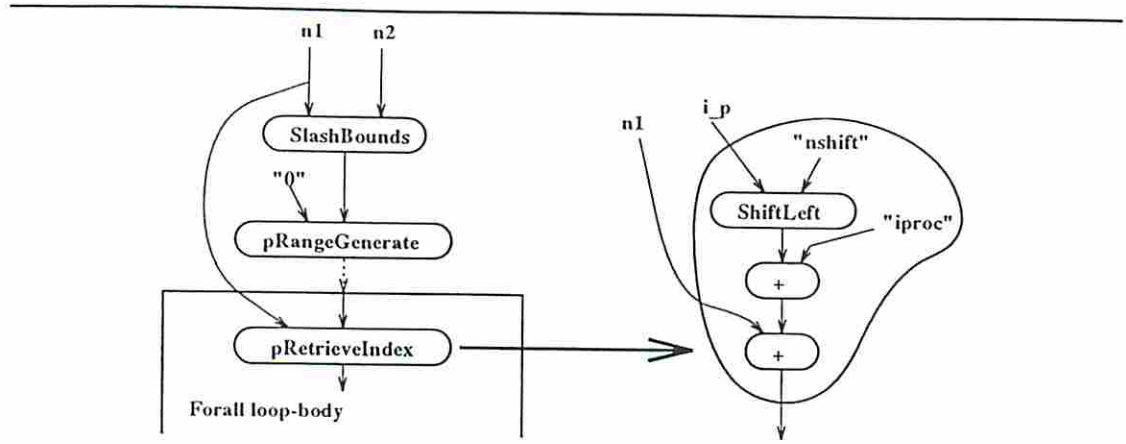


Figure 5.12: Retrieving the original index from the sliced index at the beginning of the loop-body.

Transforming *ASelect*

As we described earlier, an array must be sliced into many small pieces and each sliced block is allocated to one of the processors. Therefore, we need a mechanism to map an array index to the corresponding PE number and the index of the local array on the designated PE. Accessing an element of the array involves communication between processors (in most cases). The new construct, *pASelect* is introduced for this purpose. Indeed, *pASelect* performs the following two functions:

- To map an array index to the PE number and the index to the local array.
- Retrieve the data and copy to the local memory area.

All the *ASelect* nodes in the *forall* loop-body are replaced with *pASelect* constructs. The implementation of *pASelect* is shown in figure 5.13.

In a way similar to how we implemented the *pRetoreIndex* node, *pASelect* can also be efficiently implemented using bit-wise *shift* and *and* operations. Assuming again that the number of PEs is a power of 2, "nmask" simply becomes "nproc" 1. As can be implied by the name, the *RouterGet* construct retrieves a data

element from the designated PE using *router* communication of MP-1 [53, 54]. The cost of executing the *RouterGet* thus depends on the communication pattern. If more PEs are involved in the random array access, the overall cost of communication becomes significantly larger.

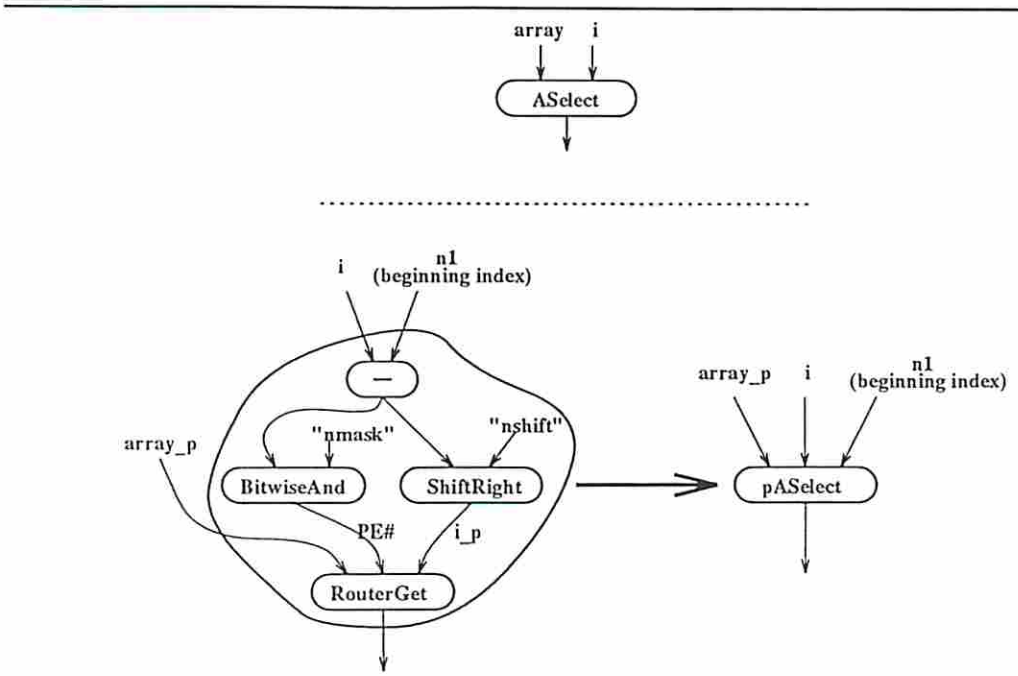


Figure 5.13: Accessing an array element across the PEs

An Example

By applying the scheme described above, the IF1 graph in figure 5.9 is transformed as shown in figure 5.14. In the figure, "v_p" is the sliced array of "v", which is assumed to be already stored in the local memory on each PE.

5.3.2 Generating MPL code

Generating MPL code from the transformed IF1 graph consists of the following steps:

- Generating the type declaration for each type of IF1 graph.
- Identifying edges which are to be implemented as plural data – In a simplistic way, all the edges in the forall body can be marked as plural data.

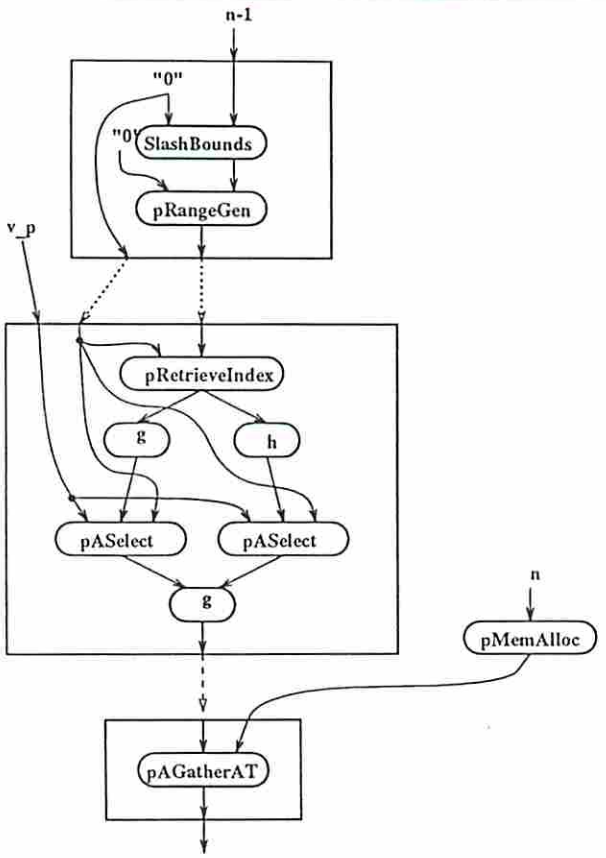


Figure 5.14: An example of transformed IF1 graph for data-parallel execution.

- Traverse the transformed IF1 graph and generate the appropriate sequence of MPL statements

Besides implementing the above steps, we also need the following for the compilation of the generated MPL code:

- Definitions of the IF1 constructs including the extended ones for data-parallelism in MPL. – This can be accomplished by macros and/or functions.

- runtime library for the execution of the compiled program. – This implements the memory allocation, input/output, and the machine specific features.

Since the actual implementation of the MPL code generation is beyond the scope of this project, we hand-coded an FFT program based on the proposed translation scheme for the validation of our work. The results are shown in the following section.

5.4 Implementation of the Target Application

5.4.1 Description of the Application – Split-Step Algorithm

Most signal processing applications are highly computation-intensive, and in many cases, real-time performance is needed. While the algorithms for solving such applications are well known and understood, their execution is still one or two orders of magnitude slower than what is needed for real-time processing. Our target application is the kernel of electro-magnetic wave propagation modeling [24]. The numerical solution of this application is centered around the *split-step* algorithm in which both forward and backward Fourier transforms are invoked repeatedly.

The *split-step* algorithm which is based on the following two equations is used to compute $u(x, z)$, the power gains (or losses) at (x,z), where x is the horizontal range and z is the vertical range from the source (*e.g.*, an antenna). Forward and backward Fourier transforms are denoted F and F^{-1} respectively.

$$U(x, p) = F[u(x, z)]$$

$$u(x+\delta x, z) = e^{i(k/2)(n'^2-1)\delta x}\ F^{-1}[U(x,p)e^{-i(p^2\delta x/2k)}]$$

where:

$$\begin{aligned} p &= k\sin\theta \\ \theta &= \text{The angle from the horizontal} \\ k &= \omega\sqrt{\mu\epsilon(a,0)} \\ \epsilon(a,0) &= \text{The permittivity just above the earth's surface} \\ a &= \text{Radius of the earth} \end{aligned}$$

The split-step algorithm is applied along the horizontal range. Therefore, the dependence between each step of computation is as follows: the label (F or F^{-1}) on the arrow shows the Fourier transform involved at each step.

$$u(0,z) \xleftarrow{F^{-1}} U(0,p) \xrightarrow{F^{-1}} u(\delta x, z) \xrightarrow{F} U(\delta x, p) \xrightarrow{F^{-1}} u(2\delta x, z) \xrightarrow{F} \cdots$$

To begin calculations, we have to find the initial value(s), $U(0,p)$. $U(0,p)$ is obtained by resolving the following equations:

$$\begin{aligned} U(0,p) &= U_e(0,p) + U_o(0,p) \\ U_e(0,p) &= 2F_c[f(z)\cos p_e z]\cos pz_A - 2iF_s[f(z)\sin p_e z]\sin pz_A \\ U_o(0,p) &= 2F_s[f(z)\sin p_e z]\cos pz_A - 2iF_c[f(z)\cos p_e z]\sin pz_A \\ f(z) &= F_c^{-1}[F_d(p)] \end{aligned}$$

Where $F_d(p)$ is the antenna pattern and F_c and F_s are cosine and sine transforms. The initial function $U(0,p)$ is obtained by proper modeling of the source which is given by $F_d(p)$ in the above equations. A detailed description of source modeling can be found in [24]. The application kernel and the recursive FFT [29, 60] are described in algorithmic form in figure 5.15 and figure 5.16 (also shown in figure 4.2), respectively.

Algorithm *split-step*

Input:

$F_d(p)$: Sampled values which represent the antenna pattern.

n : Number of steps over the horizontal range.

δx : Incremental value for each range step.

k : $\omega\sqrt{\mu\epsilon(a,0)}$ as described above.

n' : Refractive index.

θ_{max} : Maximum angle for which the antenna pattern is sampled.

n_p : Number of sampled points.

Output:

$u(x,z)$: The power gains (or losses) at (x,z)

Begin

1. Find the initial condition $U(0,p)$ from $F_d(p)$.
2. $u(0,z) \leftarrow$ *inverse_fft* $(U(0,p))$.
3. $c_1 \leftarrow e^{i(k/2)(n'^2-1)\delta x}$
4. $x \leftarrow 0, j \leftarrow 1$
5. **While** $j \leq n$ **Repeat**
   - 5-1. $c_2(p) \leftarrow e^{-i(p^2\delta x/2k)}$
   - 5-2. $u(x+\delta x,z) \leftarrow c_1$ *inverse_fft* $(U(x,p)c_2(p))$
   - 5-3. $U(x+\delta x,p) \leftarrow$ *fft* $(u(x+\delta x,z))$
   - 5-4. $x \leftarrow x+\delta x, j \leftarrow j+1$

End

Figure 5.15: The *split-step* algorithm.

Algorithm *fft*
Input:
f : array of N complex numbers, where N is a power of 2.
Output:
F : array of N complex numbers which is the Fourier transform of f.
Begin
1. Let f^e be the array of even components of f.
   Let f^o be the array of odd components of f.
2. Find $F^e = fft\ (f^e)$ and $F^o = fft\ (f^o)$.
3. Let W be $e^{2\pi i/N}$.
   For $k = 0, \ldots, N/2$, let $L_k = F_k^e + W^k F_k^o$.
   For $k = 0, \ldots, N/2$, let $U_k = F_k^e - W^k F_k^o$.
4. For $k = 0, \ldots, 2/N$, let $F_k = L_k$ and $F_{k+N/2} = U_k$. In other words, F is the concatenation of L and U.
End

Figure 5.16: The recursive FFT algorithm.

The *split-step* algorithm performs $O(n)$ Fourier transforms. The *FFT* itself has a time complexity $O(n \log n)$ when it is executed sequentially. Therefore the *split-step* algorithm takes $O(mn \log n)$ to complete its computation on a sequential machine, where m is the number of steps and n is the number of sampled points. The *Split-step* algorithm is based on iterating over each horizontal range step. In other words, each step is dependent on the result of the previous step. Therefore, the parallelism of the *split-step* algorithm is limited only by the parallelism which is achieved by the Fourier transform at each step. Thus, even if we increase the problem size with the number of range steps, parallelism does not increase. In other words, speed-up with a large number of processors can be obtained only if we have enough sampled points for the Fourier transform. Therefore, as predicted by *Amdahl's law* [2], the execution time of the algorithm is dominated by the iteration over each range step which is the sequential part of the algorithm. Assuming an infinite number of processors, the *split-step* algorithm takes $O(m \log n)$ time.

5.4.2 Analysis of the Application using SISAL Tools

Several steps are involved in developing an application in a parallel programming environment. First of all, we have to select (or design) an algorithm which is suitable for parallel implementation. The selected algorithm is then coded in a target language. The next step is to debug and optimize the program using parallel programming tools. These steps are indeed the same which are needed during the development of an application on a sequential machine. However, the appropriateness of the algorithm for parallel execution is mainly determined by the potential parallelism of the algorithm. If the algorithm itself does not have any parallelism, we cannot expect any speedup no matter the number of available processing elements. In this section, we present the simulated performance measures of the algorithm which are obtained using the SISAL programming tools.

The potential parallelism obtained by the simulation of a *split-step* with 16 sampled points over 16 range steps is shown in Figure 5.17. As we discussed in the earlier section, the *split-step* executes the loop-body sequentially over each range step. Thus, as can be seen in Figure 5.17, there is little parallelism during the execution of the whole program. However, if we look closely at the graph, we find $2n$ repeated patterns, where n is the number of range steps over which we apply the *split-step* algorithm. Parallel execution of the program, indeed, can reduce the interval between steps, since there is potential parallelism at each step, depending on the number of sampled points for which the Fourier transform is performed. The potential parallelism of a 512 point FFT is shown in Figure 5.18 (also shown in figure 4.4. The first graph shows the maximum potential parallelism with an infinite number of processors while the second part shows the clipped parallelism with 32 processors.

Running the *split-step* program for various data sizes, 64×32, 64×64, 128×32 and 128×64, where $m \times n$ means m sample points over n split-steps with 1 to 256 processors results in the theoretical speedups shown in Table 5.1 (also shown in figure 4.3) and Figure 5.19.

This simulation results confirm that a significant speedup can be obtained by employing a large number of processors only if we have enough sampled points. The fact that the number of steps over the horizontal range cannot increase the

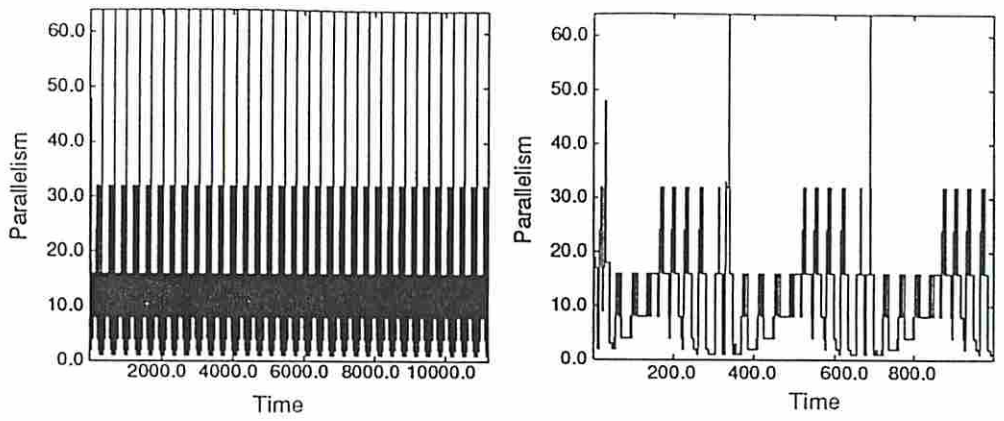
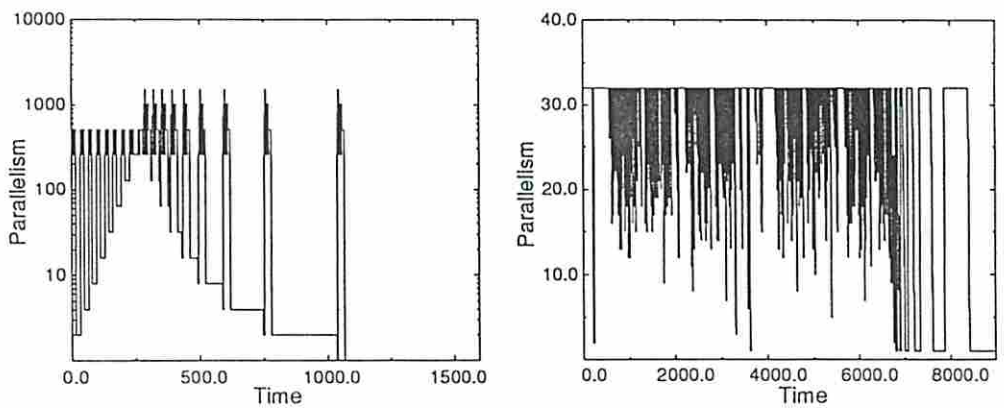


Figure 5.17: Potential parallelism of the *split-step* algorithm. The histogram shown on the right is the magnified view of the first three steps.



a) Potential parallelism of the 512pts FFT with an infinite number of PEs.

b) Potential parallelism of the 512pts FFT with 32 PEs.

Figure 5.18: Potential parallelism of FFT

	64×32	64×64	128×32	128×64
4	3.7	3.7	3.8	3.8
8	7.0	7.0	7.2	7.2
16	12.3	12.3	13.3	13.3
32	20.2	20.2	22.8	22.8
64	31.6	31.6	36.6	36.6
128	36.6	36.6	53.3	53.2
256	36.7	36.7	61.3	61.2

Table 5.1: Speedup of *split-step*.

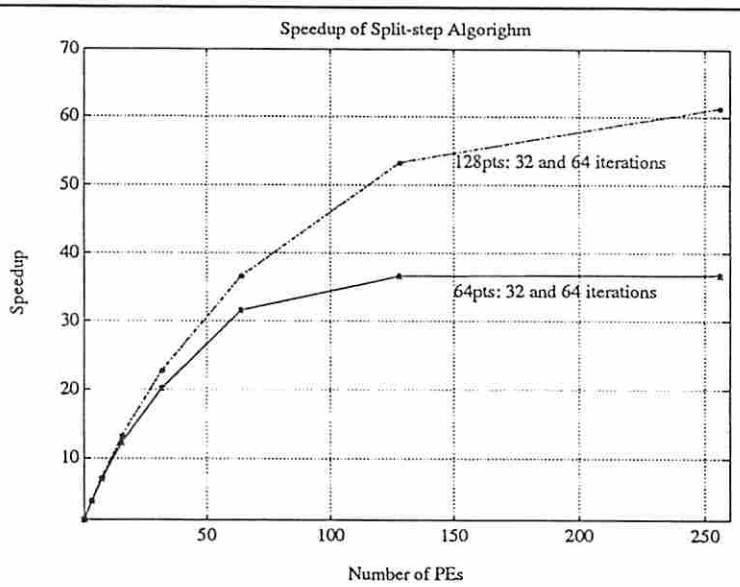


Figure 5.19: Speedup of *split-step*

potential parallelism is rather disappointing. However, we can still benefit by employing multiple processors, if there are a large number of sampled points compared to the number of available processors. For example, as can be seen in Figure 5.19, we can achieve a nearly linear speedup up to 32 processors when we have 128 sampled points in an ideal case.

5.4.3 Implementation of the Application on MP-1

As has been noted in the previous section, most of the parallelism is proportional the number of input points to the FFT function. However, the FFT algorithm in its simplest form which is recursively described (section 5.4.1) cannot be effectively implemented in parallel in a data-parallel programming paradigm. Hence, we used a different parallel FFT algorithm without any recursion.

Algorithm Description

The algorithm is based on the work shown in [34]. In the algorithm, the array index is calculated to find the *correct* data-dependency between the iterations of FFT. Let $g(l,i)$ be the i_{th} element of g which is an intermediate result after l_{th} iteration, and let $f(i)$ be the i_{th} element of input array of size $N = 2^D$, then $g(l,i)$ can be computed as follows:

$$\begin{aligned} g(0,i) &= f(i) \\ g(l+1,i) &= \begin{cases} g(l,p) + \omega_{2^l}^k g(l,q) & \text{if } 0 \le i < 2/N \\ g(l,p) - \omega_{2^l}^k g(l,q) & \text{if } 2/N \le i < N \end{cases} \end{aligned}$$

where

$$\begin{aligned} k &= \left\lfloor \frac{i}{N/2^{l+1}} \right\rfloor \\ \omega_n^k &= e^{-2k\pi i/n} \\ p &= k\frac{N}{2^l} + mod(i, \frac{N}{2^{l+1}}) \\ q &= p + \frac{N}{2^{l+1}} \end{aligned}$$

The final results are $g(D,i), 0 \le i < N$.

Note that, in the above equation, the computations of $g(l,i)$ for each $i = 0 \ldots N-1$ are independent of each other, and can therefore be computed in

parallel. Hence, if we have enough PEs, the time complexity of the algorithm based on the above equation becomes $O(D)$, *i.e.*, $O(\log N)$.

Data Mapping

The data mapping (or data partitioning) is one of the key issues concerning the performance of the program, especially in a data-parallel programming model. When mapping data into the PE array, the goals are to:

1. minimize communications
2. balance the load for greater utilization of PEs
3. keep the algorithm simple.

The first and the second issues are more performance oriented while the last one is oriented toward higher programmability. Obviously, all of the above goals cannot be simultaneously achieved (they are somewhat conflicting). Hence, we may have to give different weight to each goals when designing a data mapping strategy.

For our implementation, we chose to have the simplest data mapping scheme called *1D cut-and-stack data mapping* [53] which makes the algorithm simple. In this mapping scheme, we distribute the element of a data array over the PE array. If the number of data elements are greater than the actual size of the PE array (number of PEs), multiple data elements can be found on the same PE. The main advantage of this mapping scheme is that we can maximize the utilization of the PE array since all the data elements are evenly distributed over the PEs for greater load balancing. The actual mapping of a data element (an index in the data array) to the PEnum (*iproc* in MPL) can vary from application to application. In our implementation, we simply map i_{th} element of a data array to PE_i. Let N be the size of data array A, and z be the local array on each PE which holds a segment of the input array. Also let *nproc* be the number of PEs and *iproc* is the PE number. We first have to find out the size of local array z.

$$zs = \lceil N/nproc \rceil$$

Then for each PE,

$$z[i] = A[i \times nproc + iproc]$$

Conversely,

$$A[i] = proc[mod(i, nproc)].z[\lfloor i/nproc \rfloor]$$

Assuming that the data array is partitioned as described above and the input array is already allocated accordingly, the final data-parallel algorithm of FFT is as shown in Figure 5.20. On each PE, after the algorithm has been successfully executed, the results will still be stored in the same position as the input data element was stored.

---

for $l = 0 \ldots D-1$ do
  for $zi = 0 \ldots zs-1$ do
    $i \leftarrow zi \times nproc + iproc$
    $k \leftarrow \left\lfloor \frac{i}{N/2^{l+1}} \right\rfloor$
    $p \leftarrow k\frac{N}{2^l} + mod(i, \frac{N}{2^{l+1}})$
    $q \leftarrow p + \frac{N}{2^{l+1}}$
    $first \leftarrow$ router$[mod(p, nproc)].z[\lfloor p/nproc \rfloor]$
    $second \leftarrow$ router$[mod(q, nproc)].z[\lfloor q/nproc \rfloor]$
    if $i < N/2$ then
      $z[zi] \leftarrow first + \omega_{2^l}^k \times second$
    else
      $z[zi] \leftarrow first - \omega_{2^l}^k \times second$
    endif
  end for
end for

Figure 5.20: A data-parallel algorithm for computing FFT.

The transformation of the above algorithm for data-parallel execution (step 3 in figure 5.7) and the MPL code generation (step 4 in figure 5.7) have been manually done for our experimentation. The SISAL code of FFT and the corresponding IF1 graph are shown in the appendix. The manually coded corresponding MPL program is also shown in the appendix.

At the current state of this work, array access to other PE is implemented by using *router*[2]. The *router* implementation is the simplest way of fetching an array element from other PE, and the performance is also good when we have rather smaller size of input data. However, as the data-size gets larger, the contention on the router becomes significant and the performance is thus degraded. Further optimization may be possible in both algorithm and in translation phase. However, as shown in table 5.2 and in figure 5.21, the performance is scalable in terms of both data-size and the number of PEs.

#PE	128pts	512pts	2048pts	8192pts	32768pts	131072pts
16	0.049	0.219	1.031	4.811	—	—
32	0.028	0.116	0.526	2.438	11.146	—
64	0.021	0.062	0.270	1.236	5.634	—
128	0.017	0.042	0.174	0.786	3.560	15.970
256	0.017	0.032	0.126	0.560	2.522	11.277
1024	0.017	0.030	0.044	0.174	0.748	3.275
4096	0.017	0.030	0.037	0.071	0.280	1.173

Table 5.2: Execution time of FFT on MP-1

The execution time of the split-step algorithm based on the above FFT implementation is shown in table 5.3. In figure 5.22 and figure 5.23, we compared the execution time of split-step algorithm for various data size. The SISAL code and its hand-translated MPL code are shown in the appendix.

Since all the function calls are *side-effect-free* in SISAL, we can simply replace the function without modifying the rest of the program. There is a set of FFT functions provided by the MasPar's Math Library (MPML). These routines are highly optimized specifically for MP-1 systems. We therefore replaced the FFT

[2]In our implementation, **rfetch** routine is indeed used instead of **router** statement. – Since the array index is computed in all the PEs at the same time, the **router** statement fetches the array element whose index has actually been computed *remotely*. For example, in the statement, **router**$[p].a[i]$, i is a remote value in processor p. Therefore, using the **router** statement, we cannot access the correct remote array element whose index must be computed locally. On the contrary, **rfetch**, which is a variation of the **router** statement, uses a local array index to fetch a remote array element. An example of using **rfetch** is shown in the appendix (MPL code of FFT).

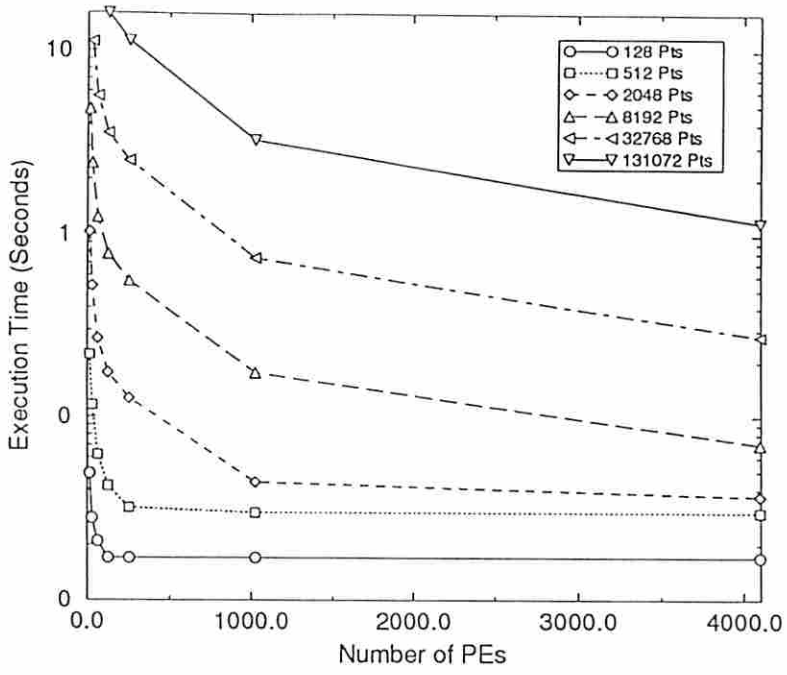


Figure 5.21: Execution time of FFT on MP-1

#PE	256×64	256×128	1024×64	1024×128	4096×64	4096×128
128	2.176	4.314	10.147	20.165	46.719	92.864
256	1.783	3.535	8.050	16.004	36.874	73.350
512	2.627	5.216	5.525	10.984	24.366	48.480
1024	2.627	5.216	2.944	5.846	12.634	25.132
2048	3.023	6.004	3.841	7.634	7.398	14.712
4096	4.183	8.315	4.842	9.628	5.871	11.677

Table 5.3: Execution time of Split-Step on MP-1. (On each column $X \times Y$ means Y iterations on X input points.)

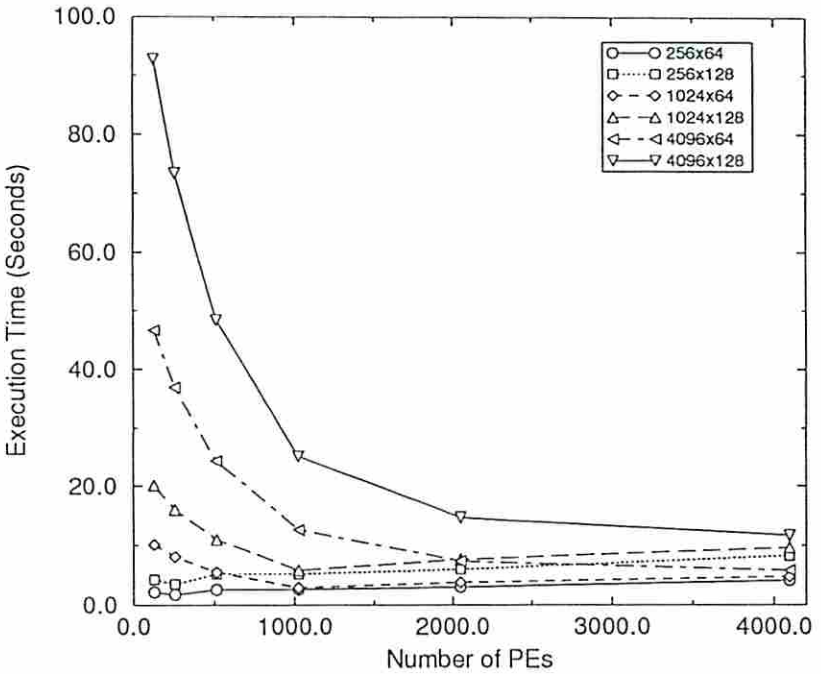


Figure 5.22: Execution time of Split-Step on MP-1

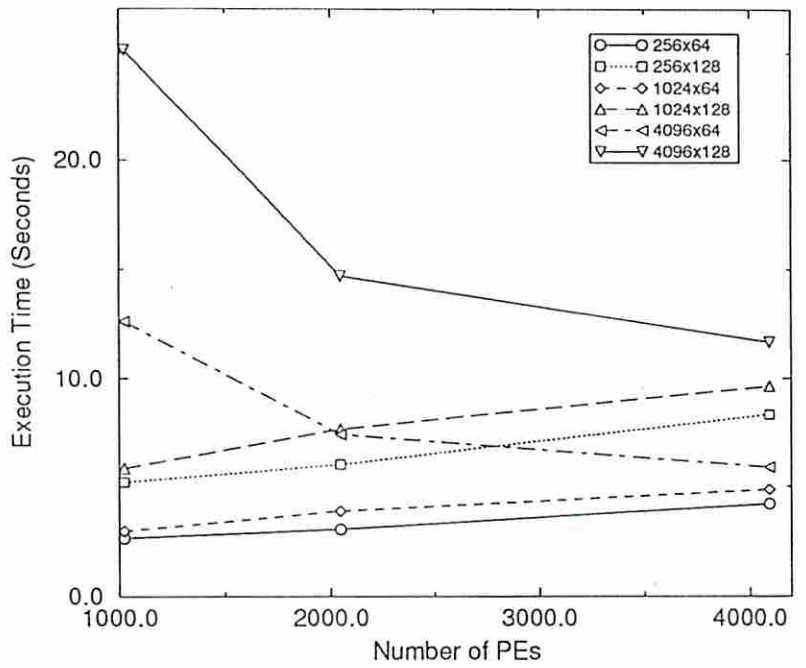


Figure 5.23: Execution time of Split-Step on MP-1: A closer look-at when the number of PEs are greater or equal to 1024.

(and inverse FFT) function calls with these MPML routines, and showed the execution time in table 5.4. This demonstrates that any big application can be highly modularized in the SISAL programming environment, and we can indeed reduce the overall development cost.

#PE	256×64	256×128	1024×64	1024×128	4096×64	4096×128
4096	0.292	0.514	0.337	0.604	0.417	0.765

Table 5.4: Execution time of Split-Step using MPML FFT library. (On each column $X \times Y$ means Y iterations on X input points.)

5.5 Discussion: SIMD *vs* MIMD

In the previous section, we have shown how a SISAL program can be implemented on the MP-1 (SIMD). Also in the our previous work [71], we presented the result of the parallel implementation of the recursive FFT algorithm on the Sequent Balance (MIMD). In this section, we thus discuss various issues of programming these two types of multiprocessor systems based on our previous work.

The most important advantage of programming in an SIMD machine is that we can exploit *massive parallelism* without paying too much control overhead. This is achieved by the synchronous execution of each instruction in SIMD model. Due to the synchronous execution, debugging can be similarly done as in a sequential machine. However, in the SIMD model, all kinds of sources of parallelism cannot be properly implemented. For example, implicit parallelism, such as double-recursion which is the core of *divide-and-conquer* style algorithm [19], cannot be effectively exploited.

On the other hand, in the MIMD model, exploitation of parallelism is not limited to the synchronous parallel execution of instructions. It is therefore possible to exploit any type of parallelism including the asynchronous parallel constructs which can enable non-strictness of a functional program. However, implementation of general parallelism is achieved at the cost of *synchronization overhead.* In MIMD model, programmers have to take care of synchronization between parallel

tasks to obtain the correct results. Implementation of this synchronization may become exponentially complex even when the parallelism increases only linearly. Hence, massive parallelism cannot be effectively handled in an MIMD machine compared to an SIMD machine.

When an application (algorithm) has highly parallel *forall* loops in its kernel and the array access pattern is regular, an SIMD machine would outperform an MIMD machine. Most signal processing applications belong to this type. On the contrary, if the algorithm mainly relies on implicit and/or dynamic parallelism, an MIMD machine could be the only choice for the parallel implementation.

There are also different issues of programming in an SIMD machine and an MIMD machine. For an SIMD machine, the main concern about parallelization is how to distribute array elements over PEs to take advantage of the regularity of an algorithm. However for an MIMD machine, programmers must pay a great deal of attention to the implementation of synchronization to obtain the correct results and also to reduce the latencies incurred by the synchronization.

A machine-independent higher level language can hide these issues by embedding them in its lower level implementation of the compiler. And the program written such a higher level language can be more portable and the development cycle can thus be reduced. For example, our application language, SISAL, proved to be the right candidate for the parallel development of various types of applications, because:

- There is no machine-dependent notation for parallel execution in the language definition.
- SISAL has already been implemented on many shared memory MIMD machines.
- In this work, we have demonstrated that a SISAL program can be also implemented on the MP-1 (SIMD machine).

The above conditions together with the general side-effect-free nature of functional languages support the SISAL as a strong candidate for the parallel implementation of computation intensive applications.

In table 5.5 and table 5.6, we presented the execution time of two different implementations of FFT. The results shown in table 5.5 are obtained by running *recursive FFT* algorithm on the Sequent Balance shared multiprocessor system. In table 5.2, we showed the execution time of data-parallel version of FFT algorithm described in section 5.4.3 on the MP-1. In both implementations, we coded the algorithm in SISAL and the the SISAL code has been translated (manually) for the C-compiler of the target machine. As can be seen on the table, we can exploit reasonable speed up on the MP-1 even when the large number of PEs are employed. However, the recursive FFT algorithm could have not been implemented effectively on the MP-1.

#PE	4096pts	16384pts	32768pts
1	40.0	214.0	501.3
2	20.4	97.0	215.6
3	14.6	66.5	142.6
4	12.6	54.7	115.8
5	10.4	51.5	102.7
6	9.9	42.9	86.7
7	8.7	42.7	82.8
8	8.5	37.1	76.6
9	8.6	33.9	73.9
10	8.0	33.5	70.4
11	8.1	32.7	70.1
12	7.6	30.3	65.7
13	7.6	30.1	65.5
14	7.5	29.4	62.2
15	7.3	29.0	62.5
16	8.4	32.3	58.7

Table 5.5: Execution time of the recursive FFT on the Sequent Balance.

#PE	128pts	512pts	2048pts	8192pts	32768pts	131072pts
16	0.049	0.219	1.031	4.811	—	—
32	0.028	0.116	0.526	2.438	11.146	—
64	0.021	0.062	0.270	1.236	5.634	—
128	0.017	0.042	0.174	0.786	3.560	15.970
256	0.017	0.032	0.126	0.560	2.522	11.277
1024	0.017	0.030	0.044	0.174	0.748	3.275
4096	0.017	0.030	0.037	0.071	0.280	1.173

Table 5.6: Execution time of the data-parallel FFT on the MP-1.

5.6 Summary

The major part of this work has been centered around the demonstration of the feasibility of employing data-parallel programming paradigm for the implementation of a signal processing application in the functional programming environment. We specifically selected SISAL as our application language mainly due to its flexibility and availability on many other shared memory multiprocessor systems. The experimental results we have obtained on the MP-1 SIMD machine have been presented together with the results on the Sequent Balance MIMD machine for the comparison between an SIMD and an MIMD machine in terms of both programmability and the performance. The summary of the work we have done in this project is as follows:

- Development of a scheme to translate a SISAL program into MPL.
- Implementation of the split-step algorithm and the FFT algorithm on the MP-1 based on the proposed translation scheme.
- Comparative analysis of SIMD and MIMD machine in terms of both programmability and performance.

As has been noted earlier, the implementation of the translator from SISAL to MPL is beyond the scope of this work. Therefore, we manually applied the

proposed translation scheme to implement our application on the MP-1. The implementation of the translator is thus planned for the future work as summarized in the following:

- Extend the translation scheme:
  - Support more SISAL constructs.
  - Improve remote array element access by analysis of the access pattern.
- Implement the translation scheme into the SISAL compiler.
- Perform more benchmarks written in SISAL on both SIMD and MIMD machines.

Chapter 6

Worker-Based Parallel Computing on Networks of Workstations

In this chapter 6, issues of parallel computing on the network of workstations are addressed in the domain of functional programming. Network-based parallel computing is becoming a potential solution for high performance computation due to its flexibility and the cost-effectiveness. Moreover, the abstract execution model described in chapter 3 particularly well fit to the network parallel computing. This chapter also describes a worker-based runtime system based on PVM (Parallel Virtual Machine) together with related issues pertinent to heterogeneous network parallel computing.

6.1 Introduction

Networks of workstations (NOW) have emerged as a cost-effective parallel computing platform [3]. A new parallel processing paradigm, *network-based computing*, has been established in the high performance computing area as well. The main constraint on obtaining performance gains with network-based computing is the communication overhead. The traditional Ethernet-based LAN (Local Area Network) which is the most popular network media provides only up to 10 to 20Mps of bandwidth. However, despite this low bandwidth compared to the interconnection network of parallel machines, for many types of applications, the parallelism can be indeed exploited and we can thus achieve some performance gains with network-based computing. Moreover, as high bandwidth LAN and WAN (Wide

Area Network) solutions emerge (figure 6.1), network-based computing is becoming more feasible than ever for high performance computing. [31]

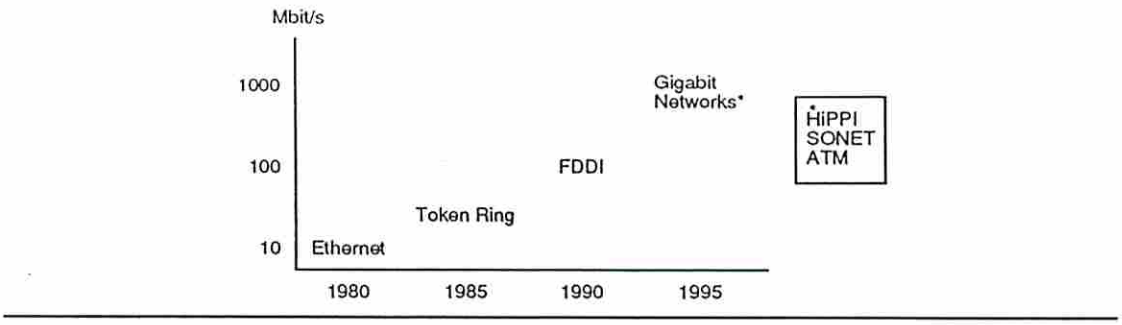


Figure 6.1: Evolution of network speed

In order for network-based computing to become a more popular parallel computing paradigm, some software solutions must be provided as well as hardware solutions. First of all, the support for parallel programming has to be implemented at the programming language level or at the operating system level. The traditional network operating system is not a natural vehicle to carry out the tasks of parallel programming. Early efforts were centered around the construction of a layer on top of an existing operating system and the definition of interfaces between the operating system and the user program for the exploitation of parallelism. The most common features which can be found in this approach are definition and/or implementation of message passing primitives together with process management primitives. With these primitives, a single user application can be distributed over the workstations and each copy of the application can synchronize with each other. PVM (Parallel Virtual Machine) [26, 31], p4 [12], and MPI (Message Passing Interface) [41] are such packages which provide libraries and/or runtime support for network-based computing.

Even though many systems already provide base solutions for network-based computing, the complexity of parallel programming based on these systems is still very high and thus may not attract many application programmers. Indeed, there are many obstacles to overcome in parallel programming:

- Parallel algorithms must be developed.

- The control of many concurrent processes (synchronization) is still too complex a task for nominal programmers.
- Even when a parallel program is developed and is proven to be functionally correct, it is not yet guaranteed to achieve performance gain.

In order to diminish the above obstacles, and thus to improve the *programmability*, we have designed and implemented a worker-based runtime model, in which a simple conventional *function call* syntax and semantics can be applied to the *parallel function call.* [16, 71] In this system, the programmer does not have to be concerned about the correctness of the program as long as its sequential execution produces the correct result. For an acceptable performance enhancement, we also developed both a compile-time and a runtime optimization scheme which works particularly well for medium to coarse grain parallelism. We chose to base our system on PVM since PVM is one of the most mature packages which also has been ported to a wide variety of platforms. The underlying parallel programming paradigm in our system is the SPMD (single program multiple data stream) model. The user application is replicated all over the machines on the network and executed asynchronously. Whenever needed, the synchronization between the copies of the application is performed using message passing primitives of PVM.

In order to further simplify parallel programming, we chose a functional language, SISAL (Streams and Iterations in a Single Assignment Language) [55], as our application language. Because a program written in a functional language is side-effect-free, the dependency between functions can only be defined as input and output parameters. In a distributed memory multiprocessing environment, message passing thus occurs only when sending and receiving function parameters. On the other hand, if we had chosen an imperative language, we would have had to pay more attention to identify dependency between functions which often occur via global variables. It could have been therefore impossible to guarantee the correctness of the parallel implementation.

The goal of this work is to provide a framework in which both high performance and the better programmability can be achieved in network computing. In the following sections, we will begin with other related works, and we will then describe

our system. The results obtained from our experimentation will be also shown in later sections.

6.2 Related Work

Since network parallel computing has been practically realized by the introduction of PVM, many other works have followed to better utilize the network computing environment. These works can be categorized as follows:

- Introduction of new network parallel computing systems.
- Resource management and task allocation.
- Scientific Supercomputing.
- Programming tools.

A number of projects based on the utilization of a collection of interconnected machines as a parallel (or concurrent) computing platform have been developed and several of them have been widely adopted. Linda [13] is a concurrent programming model based on the concept of a "tuple-space", in which cooperating processes communicate on top of a distributed shared memory abstraction. P4 [12] is a library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines. These two support both the shared-memory model and the distributed-memory model. Express [52] is a collection of tools, including message passing interface, for programming distributed memory multiprocessors and network clusters.

Resource management is an important feature that can better utilize the computing resources of the network. Since this area is applicable to distributed computing in general rather than just for parallel network computing, a few commercial products are also available as well as many research packages. The main point of these products/projects is that it is now common to have from tens to thousands of workstations on a local area network. Since they tend to be (at least temporarily) underutilized, and it should be possible to optimize the utilization for better turnaround time for user applications. Network parallel computing can also benefit these packages. More extensive review on these packages can be found in [9].

Many important scientific problems are being solved over a cluster of workstations. For example, PVM is becoming a *de facto* standard for distributed computing because of its ability to get a very good price performance ratio to solve these problems. A feasibility study of using PVM 3.0 to run *grand challenge applications* can be found in [25].

For a wider acceptability of network computing, user friendly programming tools are essential elements. Several projects have been carried out (and are currently being carried out), to enhance the parallel programming environments for network computing. HeNCE (Heterogeneous Network Computing Environment) [30] has been designed to assist scientists in developing parallel programs based on PVM. In HeNCE, an application program can be described by a graph where nodes of the graph represent subroutines and the arcs represent data dependences. The CODE model [56] takes a very similar approach but is more data-flow oriented while HeNCE programs are indeed *structured parallel flowcharts.* Another visual programming environment, VPE [27], has been developed to provide a simple human interface to the creation of message-passing programs. In VPE, a program is also represented by nodes and arcs. The major difference between HeNCE/CODE and VPE is that in VPE message passing *can* occur during the execution of the node while in both HeNCE and CODE, communication only occurs at the beginning and at the end of the node. This enables the programmer to have fine control over the communication between subroutines. We should note that the VPE model is actually a superset of HeNCE and CODE.

It should be noted that our work however emphasizes the programmability at a different level compared to the programming tools mentioned above. We try to provide compile-time and runtime mechanisms which enable the parallel execution of *existing sequential programs* with no or little modifications while most network programming tools address more of user interface for better control over the program behavior. It is therefore our belief that the combination of programming tools and some elegant compile-time/runtime mechanisms could eventually provide the ideal solution for network parallel computing.

6.3 Overview of Worker-Based Runtime System

6.3.1 SPMD Model of Execution on PVM

SPMD (single program multiple data stream) type of parallel execution can be implemented on PVM using *pvm-group* functions. In this case, all the tasks runs symmetrically (in other words, there is no master-slave relationship) on each machine (host). The PVM scheme to initialize tasks which runs in SPMD mode is described below. Note that all the tasks execute the same sequence.

1. Enroll this task in the PVM system.
2. Register this task as a member of a PVM group. If the group does not exist, the first task which tries to join the group will indeed create a new group.
3. If this task is the one which created a new PVM group, spawns new tasks. These new tasks will follow the same steps described here.
4. Wait until all the tasks reach this point (barrier synchronization).
5. Perform the main computation.

The above steps describe how a group of tasks can be initiated for the SPMD model of execution. The actual code for the above steps using PVM functions is shown in figure 6.2.

```
mytid = pvm_mytid();                /* step 1 */
me = pvm_joingroup( "foo" );        /* step 2 */

if( me == 0 )                       /* step 3 */
    pvm_spawn(argv[0], &argv[1], 0, "", NPROC-1, &tids[1]);

pvm_barrier( "foo", NPROC );        /* step 4 */

dowork( me, NPROC );                /* step 5 */
```

Figure 6.2: An example of PVM code segment to create symmetric asynchronous processes.

After the initialization of a PVM group (group "foo" in the above example), each task begins execution of the same code (in the above example, `dowork()`). In the SPMD model, however, each task can indeed execute different parts of the same program. We assume that the minimal unit of parallel execution is a *function* (*function-level parallelism*) whether it be a user-defined function or a compiler-generated function in the process of program partitioning. Parallelism is then exploited either

1. by executing a different function in another task (running on different machine)

   or

2. by executing different instantiations of the same function. (For example, we can invoke the same function with different arguments, or we can spawn multiple instances of the same loop-body for parallel loop.)

We need two basic primitives for the exploitation of parallelism:

1. ParCall : Invoke a function in parallel
2. ParJoin : Wait until the previous invoked parallel function is completed.

Besides the above two primitives, we also need to implement the parallel loop construct. The following two extra primitives are then needed:

1. LoopSpawn: Spawn multiple instances of the loop-body.
2. LoopJoin: Wait until all the instances of the loop-body are completed.

In order to implement these primitives, a runtime system must be built on top of the existing parallel execution sub-system, in our case, PVM. A worker-based runtime system has been thus designed and implemented.

6.3.2 Worker-Based Runtime System

Our *Worker-based runtime system* is another abstract layer between the parallel runtime primitives (*i.e.* ParCall and ParJoin) and the parallel execution subsystem at the lower level (figure 6.3). The detailed implementation of ParCall

and PARJOIN is hidden from the user application. In other words, in a user application, the only way to exploit the parallelism is to invoke a function using PARCALL. A PARCALL then takes care of creation of a function-task,[1] passing arguments/results and runtime scheduling/allocation.

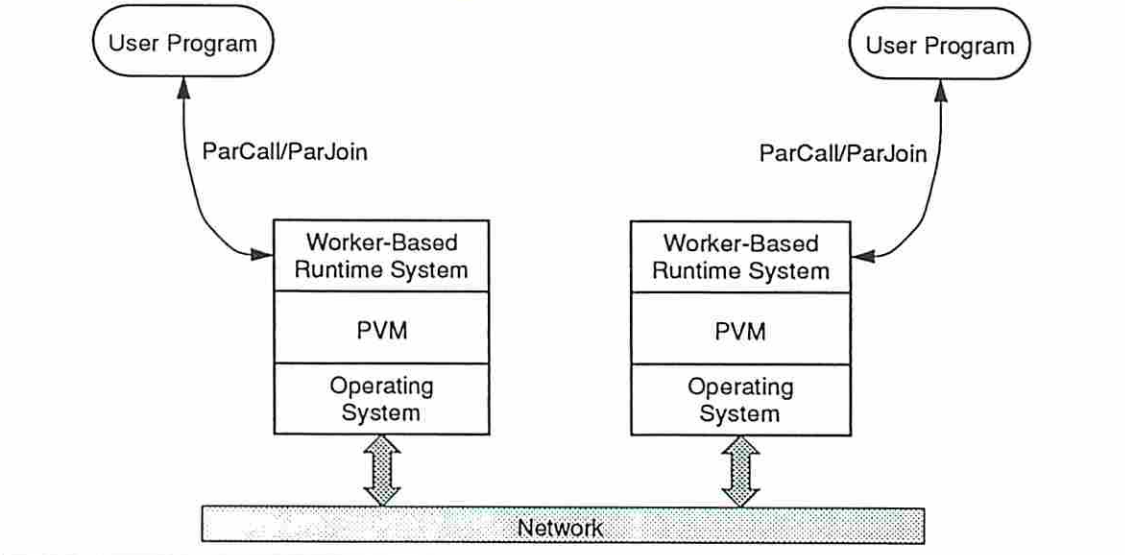


Figure 6.3: Overview of the runtime system based on PVM.

A *worker* is a process whose job is to fetch a function-task and then to execute it. Each worker process manages a single user application. This implies that if there are multiple user applications running on the PVM system, there can be several worker processes running on a single machine (each worker handles its own user application). In our implementation, a worker process is built together with the user application in a single address space. This implies that a worker process is not really a separate process. The user program and worker is put together as a single program when it is compiled for target machines. The main advantage of building a single process rather than having a worker process as a separate process is that we do not need the *intra-communication* between the worker process and the user process.

[1] Here we define two types of task, a *task of PVM* and an instance of a function which can be also called a *task* in general. We will use the term, *task* for both cases unless needed for clarification.

In order for the worker to handle tasks (which corresponds to user defined function or compiler generated function), it needs to maintain several data-structures as follows (figure 6.4):

- *Task Pool* : When a worker encounters a *ParCall*, it creates a task block and put the task block in its own *task pool*. The task pool contains the tasks which have been created so far by this worker, but not yet finished. More precisely speaking, tasks whose result is not *consumed* by this worker are stored in the task pool. Whenever *ParJoin* is called from the user program, it searches[2] for the specified task, and if the task is done, the worker passes the result to the user program and deletes the task block.

- *Ready Task Queue* : When a worker receives a request for a remote task execution, *i.e.*, receives a task block from a remote host, it stores the task block into the *ready task queue*. The worker executes the task later when there is no other preceding task. The task queue is *priority-insensitive*, which means remote tasks are executed on first-come-first-served basis.

Note that, for each task block, space for input and output is associated.

There can be two states for a worker:

- *Waiting* for any message from other workers: there are many types of messages:
  - Task Request: A task block which requests the execution of the task. The task block is followed by input arguments if any.
  - Task Completion: A task block which is finished from a remote host. The task block is followed by the output results if any.
  - Profile Message: This is for debugging/analyzing the parallel execution of the user program.

- *Running* a function (invoked by PARCALL) : The worker executes a user function and sends the results (if any) back to the originating worker. The result is then consumed by the originating worker when it executes *ParJoin*.

[2]This search operation is not indeed necessary. Since *ParCall* specifies the address of the corresponding task block, the worker can directly access the specified task block.

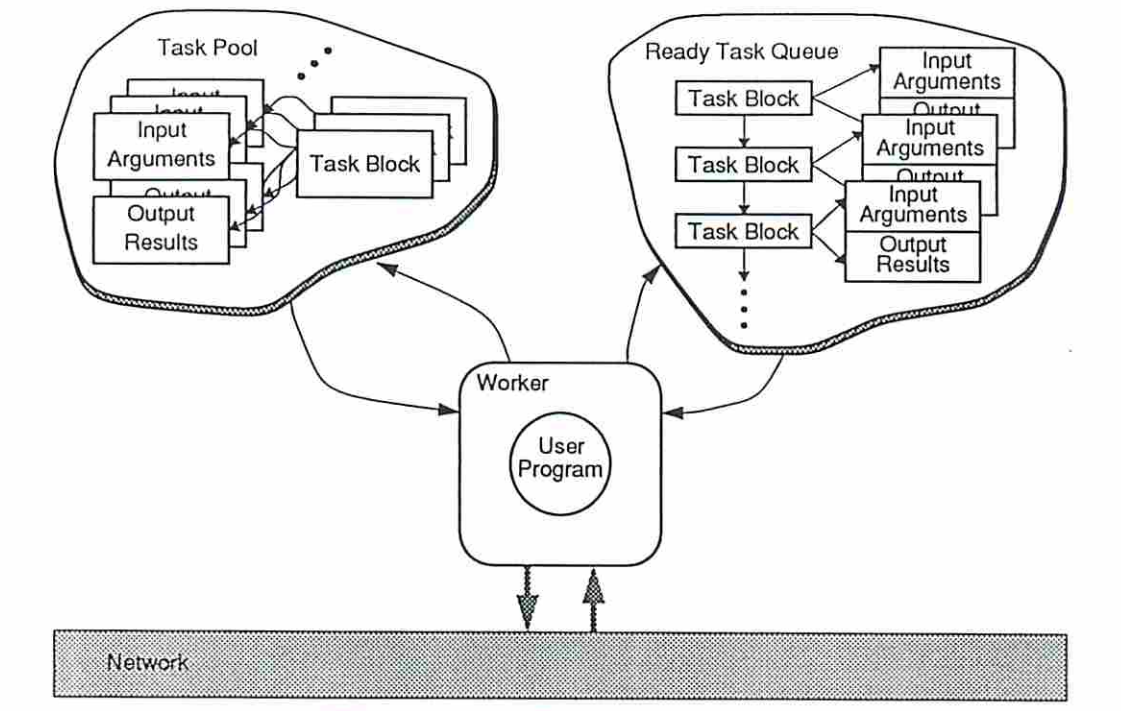


Figure 6.4: Overview of the worker on a single host

Once a user program starts execution, NPROC (number of machines) workers (tasks) are created, one for each machine. In the beginning, all the workers except the first one goes into the *waiting* state. The first worker then starts to execute the user program on its own host. Whenever PARCALL is encountered, a new task for the function is created and is sent to the other worker. The flow from the task creation to the consumption of its result is depicted in figure 6.5.

In order to implement the task-flow shown in figure 6.5, the following information is needed in the task block:

- *Task status* : One of the followings:
  - Task Request: When sending a task block to other worker.
  - Wait for Completion: Wait for the requested task is completed by the remote worker.
  - Done : The task is completed.
- *Caller*: Specifies the Worker which created this task. This field is needed when the results are sent back.

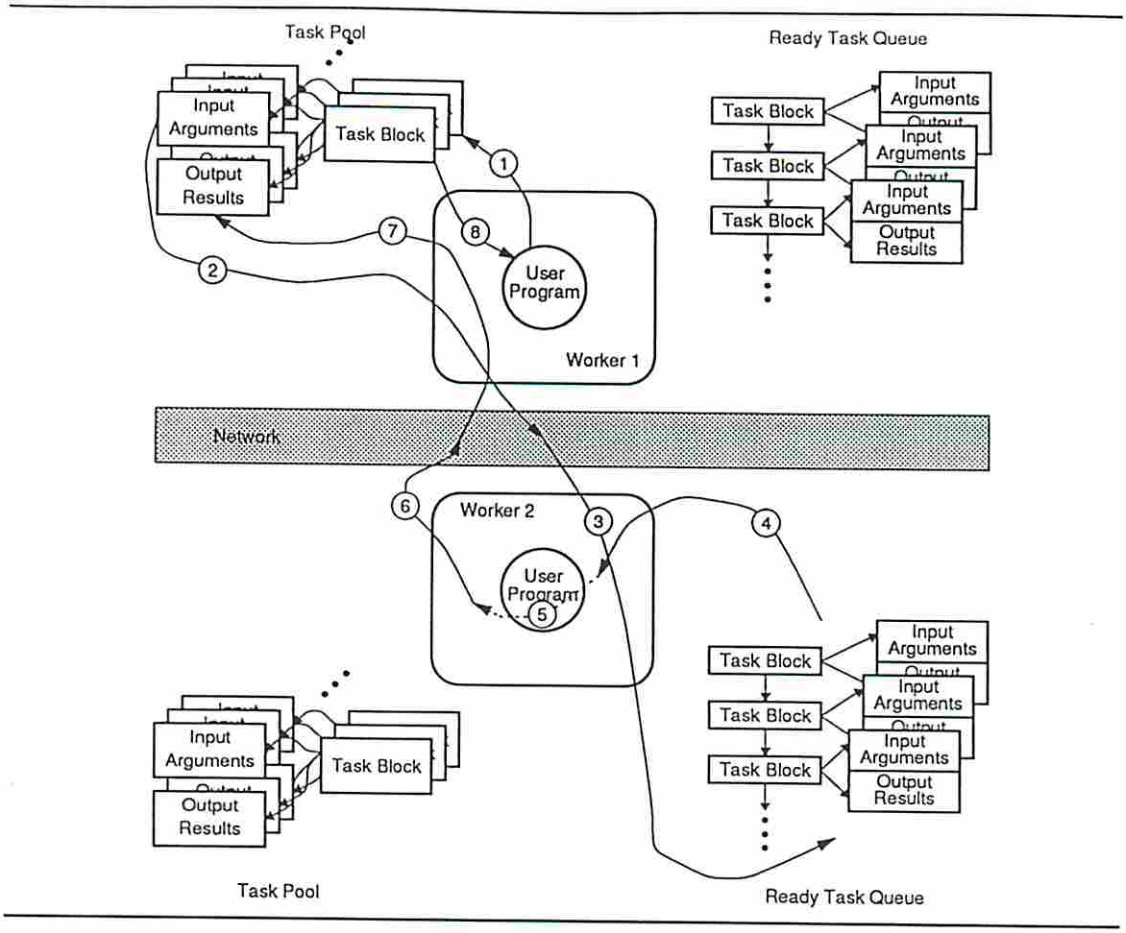


Figure 6.5: *1.* When a *ParCall* is encountered, a new task block is created. The space for its input/output arguments is also allocated. *2.* The task block and input arguments are sent to the remote worker. *3.* The remote worker (worker 2) receives the task block and input arguments. The worker then allocates the task block and space for input/output arguments in the local memory and put the task block in its local ready task queue. *4.* The worker fetches a task block along with its input arguments. *5.* Executes the corresponding function. *6.* After the completion of the function, the worker sends the task block and the results back to the originating worker. *7.* The originating worker (worker 1) receives the completed task block and results. It stores the results in the space previously allocated when the task was created. *8.* When *ParJoin* is encountered, the worker passes the result to the user program.

- *Callee*: Specifies the remote worker which will execute this task.
- *Message id* : A unique number throughout the PVM system.
- *Function id*: For each function, the compiler (or user) assigns unique id for each function. This id designates an entry in the *jump table* from which the worker finds out where to jump for the execution of the specified function.
- *A pointer to the input arguments*: For each function, all the input arguments are packed into a single structure. By packing the input arguments, it can be treated as a single message regardless of the number of arguments, which can thus reduce the message passing overhead.
- *A pointer to the results*: Like input arguments, all the results (if there are many) are packed into a single structure.

6.4 Task Allocation

The important function of *ParCall* is to select the remote worker (task allocation) where the new task is to be executed. This allocation scheme directly affects the overall performance of the worker-based runtime system. Our major goal of task allocation is to eliminate the message passing overhead while preserving the relative load balance among workers (hosts). Of course, a perfect load balancing is not possible without global resource information of the overall runtime system. However, particularly in network-computing environments, frequent message passing to maintain up-to-date global resource information can severely degrades the overall system performance. Therefore, we decided not to apply sophisticated load-balancing scheme for the allocation of the new task. Instead, we keep only a small amount of local resource information especially for the history of task allocation, and based on this information the next worker is chosen for the execution of the new task.

6.4.1 Modulo-N Allocation

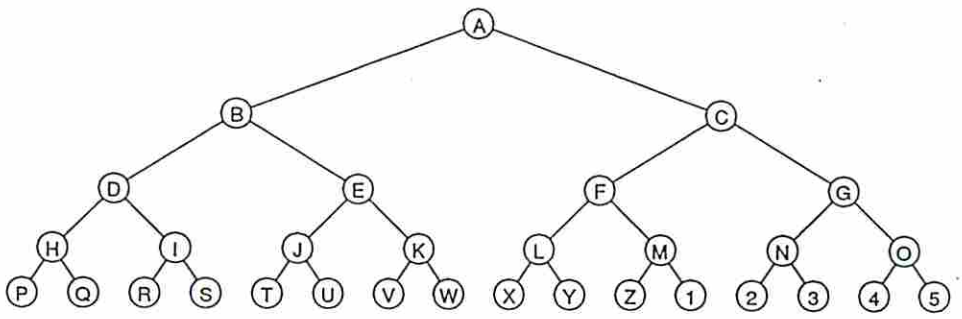
The main goal of this allocation scheme is to allocate tasks evenly throughout the workers. The basic idea is indeed similar to the *round-robin* style of task scheduling. In this allocation scheme, each host is numbered $0 \ldots N-1$ where

N is the number of hosts in the system. Then on each host h, the new task is allocated by the following rule:

1. If the first PARCALL is encountered on this host, h, allocate a new task on $(h+1)$ *mod* N.
2. If the most recently allocated machine is k, then the new task is created on $l = (k+1)$ *mod* N.

There are some cases when the next target is designated to itself. In this case, the new task is executed sequentially on the local machine exactly in the same way as the traditional sequential function call. As can be seen in the above rule, the *worker* on each host must keep track of the allocation history. The example of this allocation scheme is depicted in figure 6.6. This example is a typical form of *divide-and-conquer* style of algorithm.

<Task (Function) Call Tree>



<Allocation Map>

	Worker 0	Worker 1	Worker 2	Worker 3
Worker 0	A,O,5	B,T	C,U	N,4
Worker 1	P	Q	D,V	E,W
Worker 2	G,S	H,X	I,Y	F,R
Worker 3	J,Z	K,1	L,2	M,3
#of tasks	8	7	8	8

$Cell_{i,j} = (t_1,t_2,t_3)$ –> Worker i allocates tasks, (t_1,t_2,t_3), to Worker j

Figure 6.6: An example of Modulo-N task allocation.

As is shown in figure 6.6, each worker is assigned roughly $1/N$ of the total number of tasks, where N is the number of workers (hosts). However, note that

in this allocation we do not make any assumptions about the actual complexity of each task. In other words, even though the number of tasks handled on each worker is roughly the same, the actual amount of the work should vary. Therefore, this allocation scheme is ideal when the task partitioning is performed based on the even load-distribution. The other thing we have to point is that in the network parallel computing environment, the *capability* or the *speed* is different from worker to worker. It may thus be desirable to discriminate the low-capable worker in the allocation step. We will describe the improved but equally simple scheme, called *weighted modulo-N* in the next section.

6.4.2 Weighted Modulo-N

This scheme is the same as the *modulo-N* scheme described earlier except that a new parameter for each worker plays an important role in the allocation step. This new parameter specifies the *relative speed* of each worker (host). In our case, we assign an integer value for the speed of each worker. Fortunately, in PVM, we can statically assign the relative speed of each host. Therefore, in our worker based runtime system, we can directly import this value to determine the relative performance of each worker.

The main idea of this scheme is that the *less-capable* worker will have less chance of being assigned a new task. In order to implement this scheme as efficiently as possible, we do some preprocessing to prepare information which can be readily used in the runtime allocation step. The *allocation band* for each worker is used for this purpose. It (allocation band) is a pattern of 0's and 1's with a certain fixed length. For the less-capable worker, there are fewer 1's, *i.e.* the number of 1's represent the *speed index* of the worker. Before we go further, let us clarify two slightly different definitions of speed the index:

- *Speed Index* : Any arbitrary integer values which represents the relative speed of the worker (as we already have defined).

- *Normalized Speed Index* : The *speed index* is adjusted so that the maximum speed should be equal to the length (number of bits) of the allocation band. This value is equal to the number of 1's in the allocation band.

An example of allocation bands are shown in figure 6.7. In this example, the maximum speed index is 4 for worker 0, and the length of the allocation band is 32. Hence, the *normalized speed index* of worker 0 is 32, which is again the number of bits in the allocation band (apparently all 1's for worker 0). The number of bits of the allocation band for other workers are determined based on their speed index. In our implementation, we found that the integer value is large enough to represent the relative speed index of each worker as well as it (an integer variable) is the most efficient *unit* to handle.

Besides the number of 1's in the allocation band, we must consider the *bit pattern* of the allocation band. When the allocation band is built, the following conditions are emphasized:

- 0's and 1's have to be evenly distributed. For example, assume that worker i's turn has arrived in the *modulo-n* allocation, and also assume its (worker i) speed index is half of the maximum speed (*e.g.*, worker 1 in figure 6.7), then it may be appealing to discard the selection of the worker i every other time. The repeated pattern of "01" in the allocation band will represent this action, meaning that "if the selected worker took a new task previously, then skip the worker this time". On the other hand, if all 1's appear in the first half of the allocation band followed by all 0's in the second half, you cannot really make use of its speed index effectively, particularly if there are fewer tasks than the length of the allocation band.

- For two workers with the same speed index, the bit pattern does not have to be the same. Indeed, it is desirable to have different bit patterns. For example, assume again that two workers have speed indices of half of the maximum, and have the same pattern in their allocation bands. In this case, if one worker takes a new task when it is selected, then the other would also take a new task as well when selected in turn. However, in ideal case, it would be more desirable for those two workers to take a new task in alternating fashion.

Even though the above two conditions look contradictory to each other, they are in fact very important for the effectiveness of this *weighted modulo-N* allocation scheme.

<Example of Allocation Band for each Worker (on Worker 0)>

For Worker 0 (Speed Index 4)	11111111 11111111 11111111 11111111
For Worker 1 (Speed Index 2)	10101010 10101010 10101010 10101010
For Worker 2 (Speed Index 1)	00100010 00100010 00100010 00100010
For Worker 3 (Speed Index 3)	11111111 11111010 10101010 10101111

<Allocation Map>

	Worker 0	Worker 1	Worker 2	Worker 3
Worker 0	A,L,R,X,4	B,S,5	V	C,M,W,Y
Worker 1	2,3		D	E
Worker 2	I,O	N		H
Worker 3	F,K,T,Z	P,1	G	J,Q,U
Speed	4	2	1	3
#of tasks	13	6	3	9

$Cell_{i,j} = (t_1,t_2,t_3)$ --> Worker i allocates tasks, (t_1,t_2,t_3), to Worker j

Figure 6.7: An example of Weighted Modulo-N task allocation.

The allocation band for all the workers are built right after each worker is started. Based on this allocation band, the modified modulo-n allocation algorithm is as follows:

1. Determine the worker, say worker i, according to the original *modulo-N* scheme.
2. If the current bit-position in the corresponding (worker i's) allocation band is 1, then

   allocate a new task to worker i, and forward the current bit-position by one.

   Otherwise,

   Forward the current bit-position by one, and go over the allocation procedure(go back to step 1).

In the bottom part of figure 6.7, we showed the result of weighted modulo-n allocation for the function call tree shown in figure 6.6.

6.5 Experiment and Performance Analysis

We have run the *recursive adaptive quadrature* for the experimentation for the worker based runtime system. Both *modulo-n* and *weighted modulo-n* allocation schemes have been applied to observe the effectiveness of *weighted modulo-n* allocation scheme. In the experimentation, we chose three HP-PA workstations connected via Ethernet, each with different relative speed. The results are shown in the second part of figure 6.8. In case of all three machines are used for the parallel execution, the execution time has been reduced from 6.23 to 4.38 when the weighted modulo-N allocation scheme is applied.

The speedup can be also observed from the results. However, note that three different machines with different relative speeds have been used for the parallel execution. In other words, we can measure the speedup for each host. For example, for "Chryse", the speedup would be 16.25/4.38 = 3.71 while for "Tenedos", the speedup is 9.13/4.38 = 2.08. For the whole system, we can define the speedup,S, as follows:

$$S = min(\{S_i | S_i = speedup\ of\ host\ i\})$$

In the results shown in figure 6.8, the speedup is thus 2.08 which is the speedup of the fastest host.

In the ideal case where there is no communication cost and task creation overhead, we can formulate the speedup in terms of the relative speed of the host. First assume that there are N hosts

$$H_0, H_1, \ldots, H_{N-1},$$

with relative speed index,

$$I_0, I_1, \ldots, I_{N-1}.$$

Execution time of Recursive Adaptive Quadrature on each host for the following function:

$$f(x) = \sum_{i=0}^{N-1} x^{\frac{2i}{N}}(-1)^i$$

Host Name	Execution Time (sec)	Speed Index
Hyde	11.62	140
Tenedos	9.13	175
Chryse	16.25	100

N = 30,000, Error Tolerance = 0.00001

Execution time of Recursive Adaptive Quadrature using two allocation scheme.
(M = Modulo-N, W = Weighted Modulo-N)

Configuration	Execution Time (sec)		# Tasks on Hyde		# Tasks on Tenedos		# Tasks on Chryse	
Allocation Scheme	M	W	M	W	M	W	M	W
Hyde+Tenedos	5.28	5.03	28	25	27	30	0	0
Hyde+Chryse	8.12	7.3	28	33	0	0	27	22
Tenedos+Chryse	8.18	6.72	0	0	28	35	27	20
Hyde+Tenedos+Chryse	6.23	4.38	19	19	18	22	18	14

Figure 6.8: Comparison of execution time between modulo-n and weighted modulo-n allocation schemes.

Now we define normalized relative speed index E_i for each host H_i such that,

$$E_0 + E_1 + \cdots + E_{N-1} = 1$$

and for any two hosts H_i and H_j,

$$\frac{E_i}{E_j} = \frac{I_i}{I_j}$$

Indeed E_i can simply be calculated by,

$$E_i = \frac{I_i}{I_0 + I_1 + \cdots + I_{N-1}}$$

Then the speedup is:

$$S = \frac{1}{max(E_0, E_1, \ldots, E_{N-1})}$$

In figure 6.8,

$$E_{hyde} = 0.34, E_{tenedos} = 0.42, E_{chryse} = 0.24$$

So the ideal speedup is:

$$S = \frac{1}{E_{tenedos}} = 2.38$$

Even though our application require very little communication between workers, there is still a small difference between the ideal speedup (2.38) and the real speedup we obtained (2.08) which apparently comes from the overhead caused by the task creation and the communication. In more complex applications, the gap between the ideal speedup and the observed one may increase depending on the amount of the message passing between workers. We may have to limit the number of active tasks when the message passing overhead becomes a dominant part of the overall execution time.

6.6 Summary

Our work indends to show that network of workstations is a feasible platform for parallel computation both in terms of cost-efectivenss and programmability. We

have particularly focused on the latter issue, programmability. We have designed and implemented worker-based runtime system, in which a simple *function call* can be easily replaced with the *parallel function call.* Using our runtime system, programmers do not have to handle all the details of the parallelization, especially synchronization. Once a sequential program is proved to work correctly, its parallel implemntation using *parallel function call* should work correctly as well.

We also have presented a runtime task-allocation scheme which requires zero communiztion between the workstations while taking the capability of each workstation into acount. Our experimentation showed that this simple allocation scheme can balance the overall load among workstations reasonably well (if not optimal) based on the relative speed of each workstation. This work has indeed demonstrated that network-based computing is not only an *economic solution* for high performance computing but also *a feasible solution* in terms of programmability.

In order for this work to be more successful, there are many possible extentions to this work. As a short term project, we plan to perform more benchmarks. As a long term project the following research area can be covered:

- Full implementation of the compiler.
- Automatic program decomposition.
- More sophiticated and omproved allocation and scheduling scheme.

Chapter 7

Conclusions and Future Research

This chapter summarizes contributions of the research and suggests future research issues.

7.1 Conclusions

Today, high performance parallel computers are widely available in both industry and research fields. However, most parallel computers are used to solve only a few specific problems by using the packages which usually come together with the hardware. Indeed, development of applications on these parallel computers is by no means a feasible task for traditional application programmers due to exponentially increased complexity of programming. On the other hand, functional programming paradigm *with appropriate execution model and runtime system supports* can provide a good vehicle toward the general purpose parallel computing. The objective of this thesis is thus to provide a framework for the parallel execution of functional programs on a wide variety of platforms. An abstract execution model based on *parallel function call* has been defined to exploit parallelism at the coarse grain level. Both compile-time and runtime mechanisms have been developed to support various types of parallel architectures, including a shared memory multiprocessor system, an SIMD multiprocessor, and a cluster of networked workstations.

The contributions are summarized as follows:

- *Definition of an abstract execution model for medium-to-coarse grain parallel execution* — As we are targeting conventional parallel machines including networks of workstations, fine grained parallelism of functional programs cannot be effectively exploited. We therefore chose coarse grain approach where some significant amount of work exsists in a *grain* compared to the synchronization overhead of *parallel* tasks. Other achievements in conjunction with the definition of the model are as follows:

  - Definition of the *Control Schema* based on three basic primitives, fork, join and a regular function call.
  - Formalization of derivation from a *data-flow graph* to a *control-flow graph.*
  - Development of a scheme to optimize the control schema.
  - Definition of runtime primitives for the implementation of control schema on target systems.
  - Presentation of hierarchical partitioning as an alternative for *loop-slicing.*

- *Extension of OSC runtime system for the exploitation of dynamic parallelism for recursions* — OSC has been successful on many shared memory multiprocessors and vector processors for parallel execution of various scientific applications. However, the exploitation of parallelism was limited to a regular *forall loop* construct. We therefore extended the capability of OSC in order to take advantage of dynamic parallelism which is obtained by the *recursions.* Particularly, *divide-and-conquer* style algorithms, despite very simple notation using double recursion, have tremendous potential parallelism which grows exponentially on the size of input. By extending the OSC runtime system, we successfully achieved the performance enhancement for the applications with dynamic parallelism. There are several contributions related to this work as follows:

  - Analysis of *divide-and-conquer* algorithm in terms of exploitation of parallelism.

- Demonstration of developing parallel applications in the Sisal programming environment.
- Implementation of the runtime system for the *parallel function call* and corresponding *synchronization scheme* on a shared memory multiprocessor system (Sequent Symmetry and Sequent Balance).
- Performance analysis of the extended runtime system.

- *Development of compiling schemes of Sisal for an SIMD multiprocessor system* — Sisal has a special construct called *forall* for expressing *regular* massive parallelism. So we believed Sisal can be also effective on a platform with huge number of simple processing elements, typical SIMD multiprocessor systems. We therefore developed a compiling schemes for Sisal, more specifically, transformation mechanisms of general data-flow graphs for data-parallel programming paradigm. The other important issue was to assess the programmability and the performance aspects of SIMD type of machines as a general purpose parallel computing platforms against MIMD multiprocessors. The previous work on the Sequent Balance was a good base for comparisons. To summarize:
  - Development of transformation schemes of IF1 graph for data-parallel programming paradigm.
  - Demonstration of the parallel programming procedure (section 2.2.1) for an SIMD multiprocessor system (Maspar MP-1).
  - Comparative analysis of SIMD and MIMD machine in terms of both programmability and performance.
  - Parallel implementation of the *split-step* algorithm which is used in the modeling of electro-magnetic wave propagation.

- *Development of a worker-based runtime system for the parallel execution of functional programs on a network of workstations* — Runtime system is based on the *worker concept* used in the OSC runtime system. However, we had to apply the concept to totally different *distributed memory multiprocessing* environment. The functions of the worker therefore had to be

redefined and extended. Moreover, each host participating in the network parallel computing may be different in terms of speed or, in general, the *capability*. We have thus developed an allocation scheme called *weighted modulo-N* allocation with minimal runtime overhead. To list the related achievements:

- Design and implementation of worker based runtime system for network parallel computing paradigm : The design of the runtime system is based on the abstract execution model described in chapter 3 where actual program code and the exploitation of parallelism is *decoupled*. The worker-based runtime system can be therefore viewed as a *realization* of the *control schema*. PVM (Parallel Virtual Machine) has been used for the low level primitives such as message passing and process creation.
- Development of task allocation schemes for the network of workstations: In order to incorporate the heterogeneity of workstations, we had developed a mechanism to discriminate the *less capable* machines in the allocation step. Since, we assumed relatively low-bandwidth conventional network connection (such as Ethernet), our primary design goal was to minimize the runtime overhead. Through an experimentation, this allocation scheme has been proved to work as it had been expected with no inter-task communication overhead.
- Performance analysis: An ideal speedup of network parallel computing has been defined in terms of speed indices of participating workstations. The actual performance results have also been presented and analyzed against the ideal speedup.
- Development of a straightforward transformation scheme from the sequential function to the parallel function for the proposed runtime system: From the beginning of this work, we have the development of the compiler in mind as a future extension. Therefore, we needed to provide a transformation scheme which can be implemented without any

complex analysis of the code. As a result, we have developed a straightforward mechanism to convert any sequential function definitions/calls to the parallel function definition/calls.

7.2 Future Research Issues

While working toward the research objective stated in this thesis, many interesting points have been made. Especially, the recent work on the network of workstations demonstrated a great possibility of affordable high performance computing by effectively utilizing existing computing resources. Possible extensions to this research are described below:

- *Global load balancing* — In the current allocation scheme, load balancing is achieved without any communication overhead. The current scheme works quite well when the size of each task (amount of computation) is roughly equal. When the size of each task varies with big margin, the current scheme may not work as expected. In such situation, it may be desirable for each worker to communicate each other in order to get the *actual load* of other workers and thus put some load off from the heavily loaded workers. However, at this point it is not quite sure if this kind of *global load balancing* will indeed enhance the overall performance. The analysis of the trade-off between the current allocation scheme (with no communication overhead) and some other schemes based on global load balancing can therefore be considered as a future research.

- *Partial evaluation of functions* — The strict semantics of the sequential function call needs not be necessarily applied to the parallel function call. The higher order functions can be used according to the new language features of Sisal 2.0 [11]. In order to implement this new feature, the semantic of the parallel function call has to be changed. In other words, a new instance of the function is created even if all the input arguments are not ready. With this new approach we can also implement the multi-threaded execution model.

- *Fault tolerance* — In the network parallel computing environment, each component is an autonomous computing resource. Once any workstation crashes during the computation, other workstation can take the job. Some fault tolerant and recovery mechanisms are thus to be developed.

Some other general research issues which are also closely related to this work can be described as follows.

- *Automatic program decomposition for coarse grain parallelism* — The granularity of the program can be varied from one machine to another. In the coarse grain approach, a user defined functions or a body of the loop can serve as a good *unit* for parallel execution. However, in order to better represent the target architecture in the parallelization of an application program, there needs to be a *program decomposer* which would yield more suitable granularity for the given architecture.

- *Efficient structure management* — In a distributed memory multiprocessor environment, structure management is still one of the toughest problems. The functional semantics of structure cannot be applied *as is* due to the tremendous overhead of structure copy. Some schemes such as *update-in-place* cannot be effectively used when there is no global address space. Therefore a better scheme to reduce the message passing incurred by structure data needs to be developed.

- *Programming tools* — An integrated programming environment including coding, debugging and performance profiling tools needs to be implemented to reduce the overall cost of developing parallel applications. runtime

Bibliography

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

[2] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 483–485, Atlantic City, N.J., April 1967.

[3] T. Anderson, D. Culler, and D. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, Feb. 1995.

[4] D. Andrews and E. Barszcz. The C* parallel programming language. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, pages 93–132. North-Holland, 1992.

[5] Arvind and D. Culler. Dataflow architectures. *Annual Reviews in Compuer Science*, 1:225–253, 1986.

[6] Arvind and R. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE transactions on Computer*, 39(3):300–318, March 1990.

[7] Arvind, R. Nikhil, and K. Pingali. I-Structures: Data Structures for Parallel Computing. MIT Computation Structures Group Memo 269, Laboratory for Computer Science, MIT, Feb 1987.

[8] J. Backus. Can programming be liberated from the von Neumann style ? *Communications of the ACM*, 21(8):613–641, 1978.

[9] M. Baker, G. Fox, and H. Yau. Cluster computing review. Technical Report SCCS-748, Northeast Parallel Architectures Center, 1995.

[10] M. Beckerle. An overview of the START(*T) computer system. Technical Report MCRC-TR-28, Motorola Inc., July 1992.

[11] A. Böhm, R. Oldehoeft, D. Cann, and J. Feo. Sisal reference manual: Language version 2.0. Technical report, Computer Science Department, Colorado State University, and Computing Research Group, Lawrence Livermore National Laboratory, 1990.

[12] R. Butler and E. Lusk. Monitors, messages, and clusters: The p4 parallel programming system. Technical Report MCS-P362-0493, Argonne National Laboratory, 1993.

[13] L. Cagan and A Sherman. Linda unites network systems. *IEEE Spectrum*, December 1993.

[14] D. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.

[15] D. Cann and J. Feo. Sisal versus FORTRAN: a comparison using the Livermore loops. Technical Report (Unpublished), Lawrence Livermore National Laboratory, 1990.

[16] D. Cann, C. Lee, R Oldehoeft, and S. Skedzielewski. SISAL multiprocessing support. Technical Report UCID-21115, Lawrence Livermore National Laboratory, 1987.

[17] D. Cann and R. Oldehoeft. A guide to the optimizing Sisal compiler. Technical Report UCRL-MA-108369, Lawrence Livermore National Laboratory, Sep. 1991.

[18] U. Claussen. Parallel subpixel scanconversion. In *Proceedings of the Second Eurographics Workshop on Graphics Hardware, Amsterdam*, Spring 1988.

[19] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1990.

[20] D. Culler and G. Papadopoulos. The explicit token store. In *Journal of Parallel and Distributed Computing, Special Issue:Data-Flow Processing*, pages 289–308. Academic Press, December 1990.

[21] D. Culler, A. Sah, K. Schauser, T. Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 1991.

[22] J. Dennis. Data flow supercomputers. *IEEE Computer*, pages 48–56, November 1980.

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

[24] G. Dockery. Modeling electromagnetic wave propagation in the troposphere using the parabolic equation. *IEEE Transactions on Antennas and Propagation*, 36(10):1464–1470, October 1988.

[25] J. Dongarra, A. Geist, R Manchek, and w. Jiang. Using PVM 3.0 to run grand challenge applications on a heterogeneous network of parallel computers. In R. Sincovec *et al.*, editor, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1993. SIAM Publications.

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

[27] J. Dongarra and P. Newton. Overview of VPE: A visual environment for message-passing parallel programming. In *4th Heterogeneous Computing Workshop*, April 1995.

[28] K. Dritz. Ada solutions to the salishan problems. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, pages 9–92. North-Holland, 1992.

[29] D. Elliott and K. Ramamohan Rao. *Fast transforms: Algorithms, Analyses, Applications.* Academic Press, 1982.

[30] A. Beguelin *et al.* Graphical development tools for network-based concurrent supercomputing. In *Supercomputing '91*, pages 435–444. IEEE Press, 1991.

[31] A. Geist *et al.* . *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1994.

[32] 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.

[33] J. Feo and D. Cann. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, pages 349–366, 1990.

[34] T. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall, 1992.

[35] G. Gao, H. Hum, and J-M. Monti. Towards an efficient hybrid dataflow architecture model. In *PARLE'91*. Springer-Verlag, 1991.

[36] J-L. Gaudiot, M. Campbell, and J. Pi. Program graph allocation in distributed multicomputers. *Parallel Computing*, 7(2), June 1988.

[37] J-L. Gaudiot and M. Ercegovac. Performance evaluation of a simulated dataflow computer with low-resolution actors. *Journal of Parallel and Distributed Computing*, 1985.

[38] J-L. Gaudiot and C. Kim. Data-driven and multithreaded architecture for high-performance computing. In T. Casavant, P. Tvrdík, and F. Plášil, editors, *Parallel Computers: Theory and Practice*, chapter 2, pages 73–106. IEEE Computer Society Press, 1996.

[39] J-L. Gaudiot and L. Lee. Occamflow: A methodology for programming multiprocessor systems. *Journal of Parallel and Distributed Computing*, August 1989.

[40] J-L. Gaudiot and D-K. Yoon. Occam. In J. Feo, editor, *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, pages 217–262. North-Holland, 1992.

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

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

[43] 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.

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

[45] 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.

[46] 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.

[47] P. Hudak and B. Goldberg. Serial combinators: Optimal grains for parallelism. In Jean-Pierre Jouannaud, editor, *Functional Programming languages and Computer Architecture*, pages 382–399. Lecture Notes in Computer Science 201, Springer-Verlag, 1985.

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

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

[50] 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.

[51] D. Knuth. *Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.

[52] A. Kolawa. The express programming environment. In *Workshop on Heterogeneous Netowrk-Based Concurrent Computing*, Tallahassee, October 1991.

[53] MasPar Computer Corporation. *Data-parallel Programming Guide*, 1991.

[54] MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) User Guide*, 1992.

[55] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL : Streams and Iteration in a Single Assignment Language – language reference manual version 1.2. Technical Report TR M-146, Lawrence Livermore Laborartory, March 1985.

[56] P. Newton and J. Browne. The CODE 2.0 graphical parallel programming language. In *ACM Int. Conf. on Supercomputing*, July 1992.

[57] R. Nikhil and Arvind. Can dataflow subsume von neumann computing? In *Proceedings of the* 16^{th} *Annual International Symposium on Computer Architecture*, pages 262–272. IEEE, 1989.

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

[59] A. Osterhaug, editor. *Guide to parallel programming*. Prentice Hall, 1989.

[60] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical Recipes.* Cambridge University Press, 1986.

[61] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[62] V. Sarkar and D. Cann. POSC — a partitioning and optimizing Sisal compiler. Technical report, Lawrence Livermore National Lab., 1990.

[63] V. Sarkar and J. Henessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 17–26, July 1986.

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

[65] S. Skedzielewski and J. Glauert. IF1: An intermediate form for applicative languages reference manual, version 1.0. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.

[66] V. Srini. An architectural coomparison of data-flow systems. *IEEE Computer*, pages 68–88, March 1986.

[67] P. Treleaven, D. Brownbridge, and R. Hopkins. Data-driven and demand-driven computer architecture. *ACM Computer Surveys*, 14(1), March 1982.

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

[69] M. Welcome, S. Skedzielewski, R. Yates, and J. Ranelleti. IF2: An applicative language intermediate form with explicit memory management. University of California Lawrence Livermore National Laboratory, Manual M-195, November 1986.

[70] D-K. Yoon and J-L. Gaudiot. Worker-based parallel computing on pvm. submitted to *EUROPAR 96.*

[71] D-K. Yoon and J-L. Gaudiot. Programming and evaluating the performance of signal processing applications in the sisal programming environment. In *Proceedings of the Second Sisal Users' Conference*, pages 67–82. Lawrence Livermore National Laboratory, 1992.

Appendix A

Implementation of Split-Step Algorithm on MP-1

A.1 SISAL code of Split-Step

```
%$entry=SplitStep
define SplitStep

%
%               FAST FOURIER TRANSFORM (Recursive)
%

type complex =          record [r, i:double_real];
type complexOneDim =    array [complex];
        % Index ranges from 0 to N

%
%
% Implementation of Split-Step algorithm in SISAL.
%
% In this implementation, U(0,p) is given as an initial value.  U (0,p)
% is actually obtained by the apperture function.  Apperture function is
% again obtained by the antenna pattern.  In the following function, we
% start from this U (0,p) and calculate u(x,z) by iteration.
%
% The solution of this function covers the horizontal range from 0 to
% n x dx.  Other parameters, k and nprime, are just assummed to be a
% constant values in this implementation.
%

function SplitStep (
     Uzero:complexOneDim; % Initial solution
     n:integer;           % Number of iterations.
```

```
     dx:double_real;              % incremental range step
     k:double_real;               %
     nprime:double_real;          % refractive index
     thetamax:double_real;        % Maximum angle for which antenna
                                  % pattern is sampled
     nsample:integer      % number of sampled points
returns
     array [complexOneDim])

     let
          dtheta := thetamax / double_real (nsample - 1);

          % prange is the array of p over which the antenna pattern
          % is sampled
          prange := for i in 0,nsample-1
                    theta := dtheta * double_real (i);
                    p := k * sin (theta);
               returns
                    array of p
               end for;
          t1 := (k/2.0d)*(nprime*nprime -1.0d)*dx;
          coef1 := record complex [r:cos(t1); i:sin(t1)];
     in
          for initial
               i := 1;
               bigU := Uzero;
               u := ifft (bigU);
          while i <= n
          repeat
               t2 := for j in 0,nsample-1
                     returns
                          array of prange[j]*prange[j]*dx / (k*2.0d)
                     end for;
               coef2 := for j in 0, nsample-1
                     returns
                          array of record
                             complex [r:cos(t2[j]); i: -sin(t2[j])]
                     end for;
               newU := for j in 0, nsample-1
                     returns
                          array of mulc(old bigU [j],coef2[j])
                     end for;

               tu := ifft (newU);
```

```
                u := for j in 0,nsample-1
                     returns
                          array of mulc (coef1,tu[j])
                     end for;
                bigU := fft (u);
                i := old i + 1;
           returns
                array of u
           end for
      end let

end function
```

A.2 MPL code of Split-Step

```
#include "common.h"


void SplitStep (bigU, u, zs, n_iter, dx, k, nprime, thetamax, n_points)
  plural complex_t *bigU; /* Input complex numbers: initial solution */
  plural complex_t *u;  /* The solution */
  int   zs;             /* Size of bigU[] and u[] */
  int   n_iter;         /* Number of iterations */
  int   n_points;       /* Number of sampled points */
  real  dx;             /* Incremental range step */
  real  k;              /* */
  real  nprime;         /* Refractive index */
  real  thetamax;       /* Maximum angle for which antenna pattern */
                        /* is sampled */
{

   int depth;                    /* log(n_points) */
   plural int **j1;              /* List of j1 values. */
                                 /* size is zs*depth  */
   plural int **j2;              /* List of j2 values */
   plural complex_t **zomega;    /* List of omega values */

   /* Misc. scratch arrays and variables */
   int i;
   real t1;
   plural real p_t2;
   plural complex_t *work1;
   plural complex_t *work2;
   plural int p_i,p_idx;
```

```
/* constants which computed once in the beginning of the function */
plural real *prange;
real dtheta;
complex_t coef1;

/***************************/
/* Allocate scratch arrays */
/***************************/

prange = (plural real *) p_malloc (sizeof(real)*zs);
work1 = (plural complex_t *) p_malloc (sizeof(complex_t)*zs);
work2 = (plural complex_t *) p_malloc (sizeof(complex_t)*zs);


/************************************/
/* Allocate various tables for FFT  */
/************************************/

depth = ilog2(n_points);

j1 = (plural int **) p_malloc (sizeof(int *)*zs);
                              /* Index lookup tbl 1 */
j2 = (plural int **) p_malloc (sizeof(int *)*zs);
                              /* Index lookup tbl 2 */
zomega = (plural complex_t **)
   p_malloc (sizeof(complex_t *)*zs); /* Table of roots on unity */

for (p_i=0;p_i<zs;p_i++) {
   j1[p_i] = (plural int * plural) p_malloc (sizeof(int)*depth);
   j2[p_i] = (plural int * plural) p_malloc (sizeof(int)*depth);
   zomega[p_i] = (plural complex_t * plural)
      p_malloc (sizeof(complex_t)*depth);
}


/* Calculate the intervals over which data are sampled */

dtheta = thetamax / ((real) (n_points - 1));
for (p_i=0; p_i<zs;p_i++) {
   plural real p_theta;

   p_idx = (p_i<<MemShift) + iproc;
   p_theta = dtheta * (plural real) p_idx;
```

```
   prange[p_i] = k  * fp_sin (p_theta);
}
t1 = (k/2.0)*(nprime*nprime - 1.0)*dx;
makec (f_cos(t1), f_sin(t1), coef1);


/* Prepare for the FFTs */
fft_init (j1,j2,zomega,zs,depth, n_points);

/* perform initial fft (inverse fft ) */

for (p_i=0;p_i < zs ; p_i++) {
   copyc (bigU[p_i], u[p_i]);
}

/* The actual solution of SplitStep() is the *array* of "u", */
/* i.e., the solution is two dimensional array which contains*/
/* all the values over the iteration steps.  For the sake of */
/* simplicity and the memory consideration, we only returns */
/* the last element of array of "u". */

ifft (u,work1,j1,j2,zomega,zs,depth,n_points);
                                /* the first solution */


/*********************************/
/* Now we begin the main  loop  */
/*********************************/

for (i=0;i<n_iter;i++) {

   for (p_i=0;p_i<zs;p_i++) {
      p_t2 = prange[p_i] * prange[p_i] * dx / (k*2.0);
      makec (fp_cos (p_t2), -fp_sin (p_t2), work1[p_i]);
      mulc (bigU[p_i], work1[p_i], work2[p_i]);
   }

   ifft (work2,work1,j1,j2,zomega,zs,depth,n_points);

   for (p_i=0;p_i<zs;p_i++) {
      mulc (coef1, work2[p_i], u[p_i]);
   }

   for (p_i=0;p_i<zs;p_i++) {
```

```
            copyc (u[p_i], bigU[p_i]);
        }

        fft (bigU,work1,j1,j2,zomega,zs,depth,n_points);
    }
}
```

A.3 SISAL code of FFT

```
%$entry=fft
define fft

%
%               FAST FOURIER TRANSFORM (Non-Recursive Version)
%

type complex =          record [r, i:double_real];
type complexOneDim =    array [complex];
       % Index ranges from 0 to N

global sin(x : double_real returns double_real)
global cos(x : double_real returns double_real)

function addc(a, b:complex returns complex)
     record complex [ r:a.r+b.r ; i:a.i+b.i ]
end function % addc

function subc(a, b:complex returns complex)
     record complex [ r:a.r-b.r ; i:a.i-b.i ]
end function % subc

function mulc(a, b:complex returns complex)
     record complex [ r:a.r*b.r-a.i*b.i ; i:a.i*b.r+a.r*b.i ]
end function % mulc

function makec(a, b:double_real returns complex)
     record complex [r:a; i:b]
end function

function ilog2(n:integer returns integer)
     for initial
          x := -1;
          k := n;
     while k > 0
```

```
        repeat
            k := old k / 2;
            x := old x + 1;
        returns
            value of x
        end for
end function

%
% fft : non-recursive fast fourier transform.
%
% input
%       v : Array of Complex values
%
% output
%       fft of v
%

function fft(v : complexOneDim returns complexOneDim)
    let
        PI := 3.1415926536D;
        twoPI := 2.0D*PI;
        n := array_size (v);
        half_n := n/2;
        depth := ilog2 (n);
    in
        for initial
            res := v;
            l := 0;
            n1 := 2;
            n2 := n/2;
            n3 := n;
        while l < depth
        repeat
            theta := twoPI / double_real (old n1);

            res := for j3 in 0,n-1
                       t3 := if j3 < half_n
                             then j3
                             else j3 - half_n
                       end if;
                       k := t3 / old n2;
                       i := mod (t3, old n2);
                       j1 := k * old n3 + i;
```

```
                    j2 := j1 + old n2;

                    theta_k := theta * double_real (k);

                    omega := makec (cos (theta_k),-sin (theta_k));
                    omega_f := mulc (omega, old res[j2]);
                    myres := if j3 < half_n then
                                addc (old res[j1], omega_f)
                             else
                                subc (old res[j1], omega_f)
                             end if
                returns
                    array of myres
                end for;
            n1 := old n1 * 2;
            n2 := old n2 / 2;
            n3 := old n3 / 2;
            l := old l + 1;
        returns
            value of res
        end for
    end let
end function
```

A.4 IF1 Graph of FFT

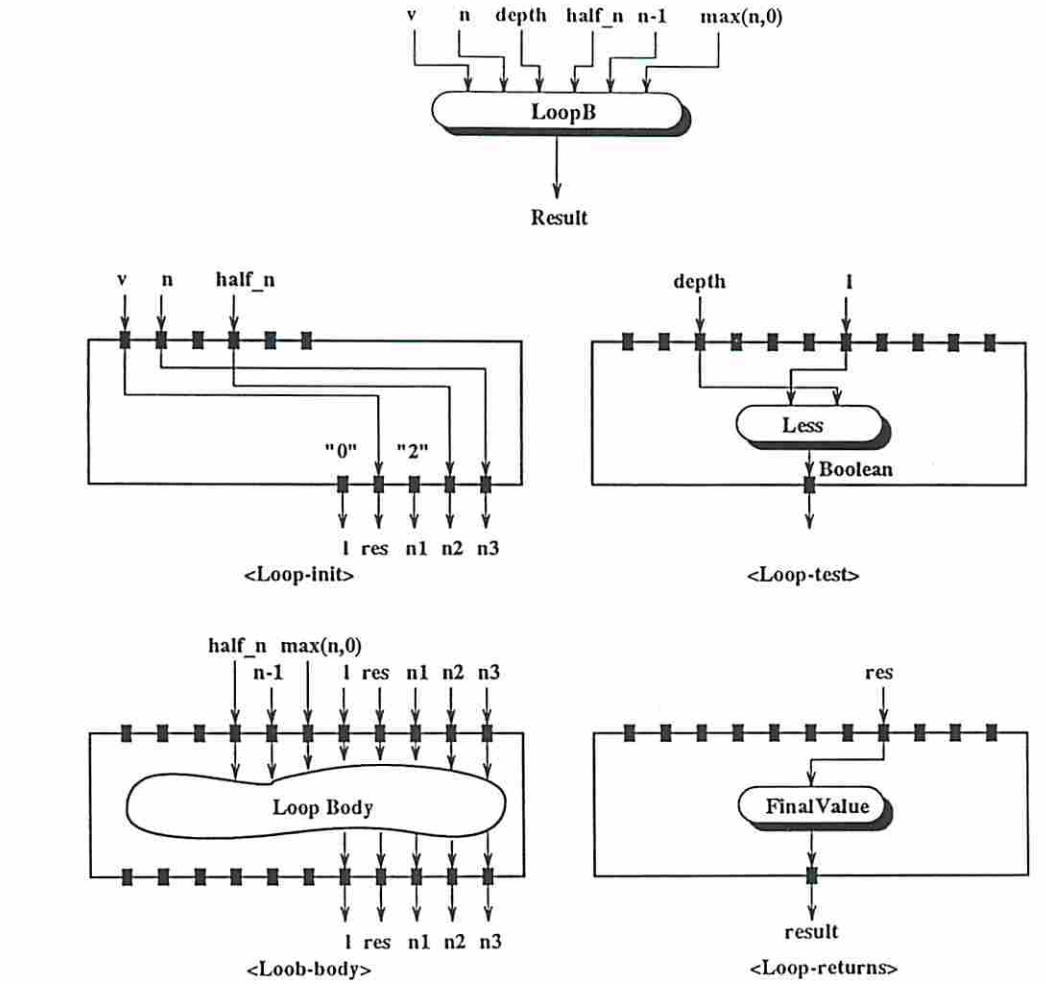


Figure A.1: IF1 graph of LoopB.

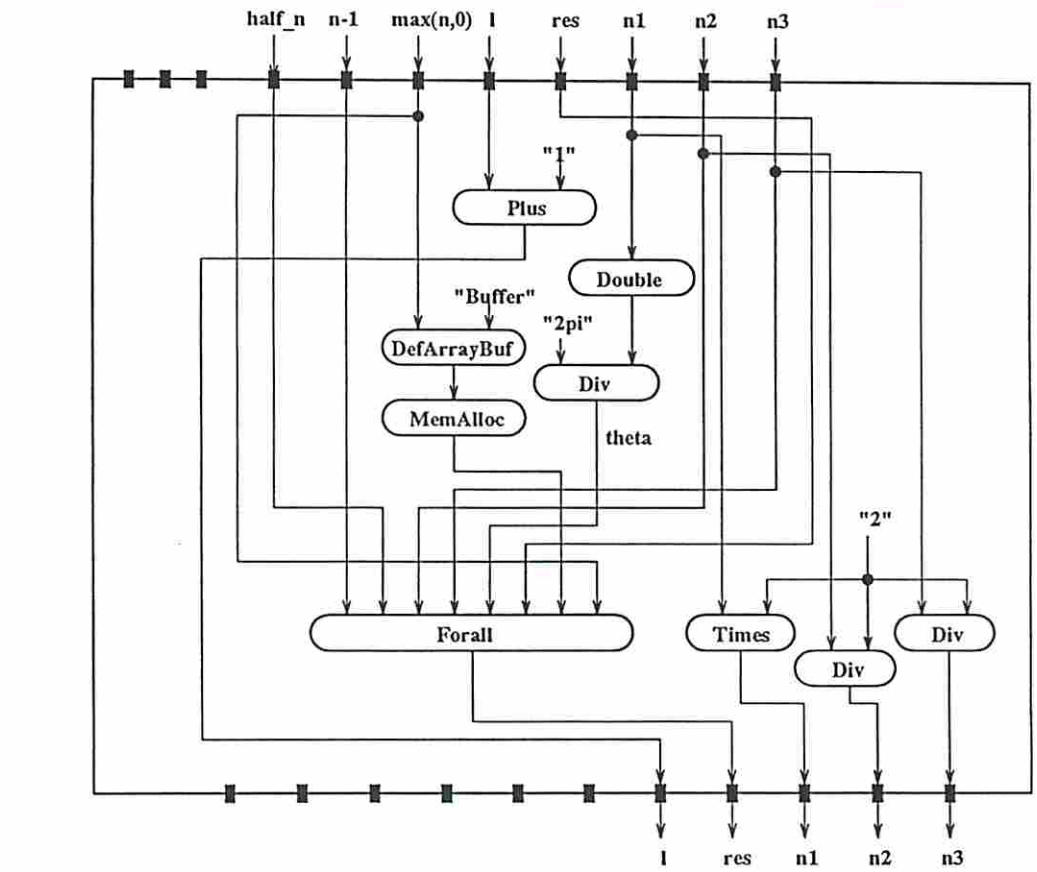


Figure A.2: IF1 graph of LoopB-body.

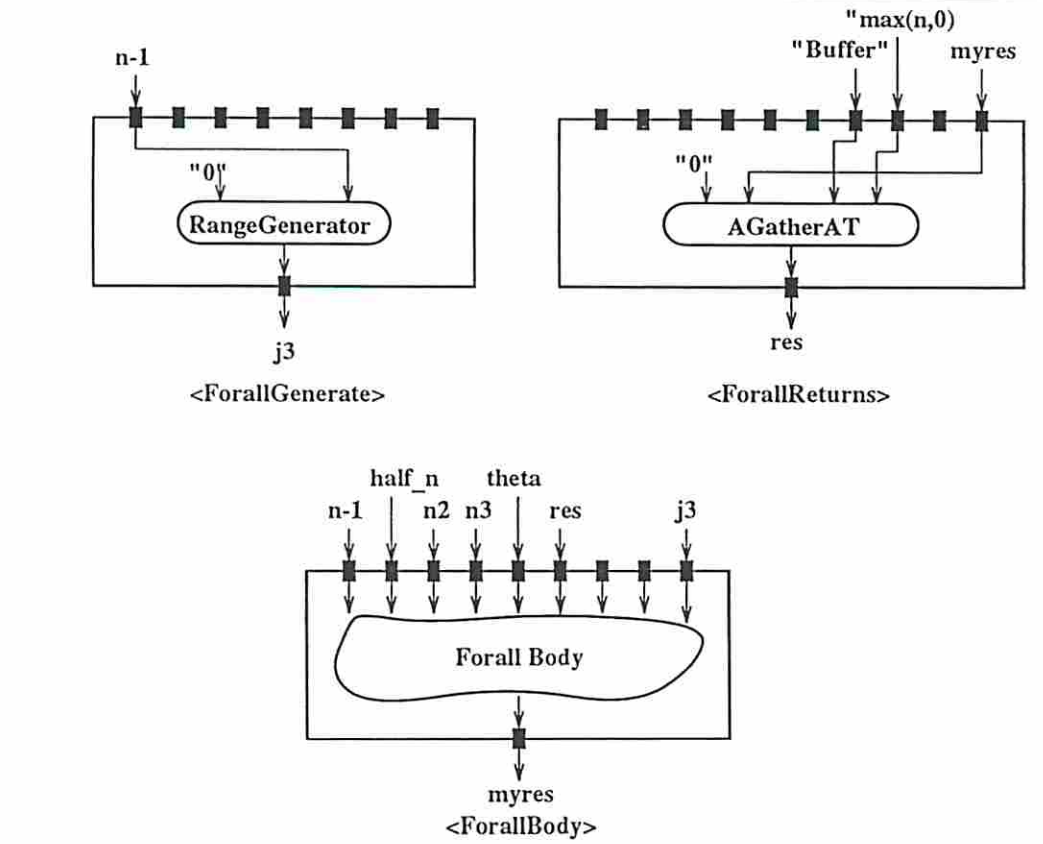


Figure A.3: IF1 graph of Forall.

A.5 MPL code of FFT

```
main()
{
   ...
   nmask = NumPEs-1;
   nshift = ilog2(NumPEs);
   ...

   /* Prepare input on all the PEs */
   zs = (int) ceil ((double) Asize / (double) NumPEs);
   z = (plural complex_t *) p_malloc (sizeof(complex_t)*zs); /*In*/
   newz = (plural complex_t *) p_malloc (sizeof(complex_t)*zs);/*Out*/
   for (p_i=0;p_i<zs;p_i++) {
      plural int p_idx;

      p_idx = (p_i<<nshift) + iproc;
      if (p_idx == 0 || p_idx >= (Asize-1))
          makec (0.0,0.0,z[p_i]);
      else
          makec (1.0,0.0,z[p_i]);
   }

   ...
   fft (z, newz, Asize, zs); /* Call fft */
   ...

}


void fft (a,b,n,zs)
  plural complex_t *a,*b;
  int n,zs;
{
   plural complex_t p_omega,p_omega_f,p_lc,p_rc;
   plural real p_t,p_theta_k;
   plural int p_k,p_i,p_zi;
   plural int p_j1,p_j2,p_j3; /* f(j3,l+1) = f(j1,l) +(-) w*f(j2,l) */

   real twoPI,theta;
   int half_n = n/2;
   int depth,l; /* depth log2(n) */
   int n1,n2,n3; /* 2^x, used in various ways. See below */
```

```
  twoPI = PI*2.0;
  depth = ilog2(n);
  n1 = 2; /* 2^(l+1) */
  n2 = n/2; /* N/(2^(l+1)) */
  n3 = n; /* N/(2^l) */

  for (l=0;l<depth;l++){
     theta = twoPI / n1; /* for w(2^(l+1),x) */

     for (p_zi=0;p_zi<zs;p_zi++){
plural int p_tj3;

p_j3 = (p_zi<<MemShift) + iproc;
 /* p_j3 the index this PE is procsessing */
if (p_j3 < half_n)
   p_tj3 = p_j3;
else
   p_tj3 = p_j3 - half_n;

p_k = p_tj3/n2;
p_i = p_tj3%n2;
p_j1 = p_k*n3 + p_i;
p_j2 = p_j1 + n2;

/* use "rfetch" for remote array access */
ps_rfetch (p_j1&ProcMask, (plural char * plural)&(a[p_j1>>MemShift]),
   (plural char *)&p_lc, sizeof (complex_t));
ps_rfetch (p_j2&ProcMask, (plural char * plural)&(a[p_j2>>MemShift]),
   (plural char *)&p_rc, sizeof (complex_t));

p_theta_k = theta * p_k;
makec (p_cos(p_theta_k), -p_sin(p_theta_k), p_omega);

mulc (p_omega, p_rc, p_omega_f);
if (p_j3 < half_n)
   addc (p_lc,p_omega_f,b[p_zi]);
else
   subc (p_lc,p_omega_f,b[p_zi]);
     }

     n1 *= 2;
     n2 /= 2;
     n3 /= 2;
```

```
      /* copy back the results for next iteration */
      for (p_zi=0;p_zi<zs;p_zi++)
 copyc (b[p_zi],a[p_zi]);
   }
}
```