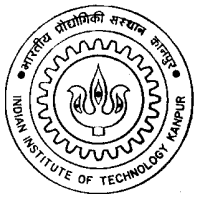
A Parallelizing Compiler for a Cluster of Workstations

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology

by

Yatin Nayak



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

April, 2000

Certificate

This is to certify that the work contained in the thesis entitled " *A Parallelizing Compiler for a Cluster of Workstations*", by *Yatin Nayak*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

April, 2000

(Dr. Sanjeev Kumar Aggarwal)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Acknowledgements

I wish to take this opportunity to express my sincere and heartfelt gratitude to my thesis supervisor, Dr. Sanjeev Kumar Aggarwal, for providing me with invaluable guidance and support throughout this thesis work. His encouraging words in the face of seemingly unending obstacles always boosted my sagging spirits and motivated me to keep going.

I would also like to thank Dr. Rajat Moona for creating in me the desire to learn new things. I have always been awed by his deep knowledge in almost every sphere and I hope I can come close to emulating him one fine day. My whole-hearted thanks to the faculty members of Computer Science and Engineering department for imparting me with invaluable knowledge and adding a lot of value-oriented growth to my career. Thanks are also due to the technical staff for providing an excellent working environment.

It was a whole new experience for me at IIT Kanpur and I enjoyed every minute of it. I shall never forget the friends I have made here, each one of them special to me in their own right. Venky was a constant source of inspiration and entertainment for me, and I never got bored of his unending chatter. I enjoyed being in the company of stalwarts like rahulj and Godha, and I learnt a lot from them. Pogde was a great companion to be with, and I shall surely miss my daily tea-breaks with him and Rajiva. And how can I forget Khannaji, Mujoo, Unmesh, Pinaki, Ghulam, Ramachandra and Prakash for the lovely companionship they provided me with. I also owe my thanks to the SUIF mailing list for happily accepting me into the SUIF world and helping me sort out numerous problems. In particular, I would like to acknowledge the unflinching and kind help provided by Brian Murphy and David Heine, research students at Stanford University, and Xavier Verians, a Ph.D. student at the Universite Catholique de Louvain in Belgium.

My parents, my brothers Amit and Anay and cousin Priya have been a constant

source of love and affection throughout my life and needless to say, without their love and support, I would never have been able to make it into such a premier institute of learning as IIT Kanpur. I owe my success to them for everything. Finally, this thesis would have been incomplete without Deepa, who provided me with invaluable emotional support and encouragement to carry on in the face of all odds. She has been a great source of inspiration and motivation.

Yatin Nayak

Abstract

Clusters of workstations are rapidly becoming the order of the day for supporting high-performance, computationally-intensive parallel applications in the field of parallel computing. Their performance is on the rise and they provide a cheaper and more viable alternative to multiprocessor systems and supercomputers. A cluster of workstations provides a *global single system view*, it can be thought of as a *virtual machine*, a cluster of powerful desktops connected by a high-speed network, that logically appears to the user as a single large parallel computer. They have finally begun to move from laboratories in the academic and scientific community to the realms of commercial computing. This move has been fueled by ever-advancing technology, spiraling costs and an insatiable hunger for more computing power. Current clusters are based on the message passing model of parallel computing in which programmers deal explicitly with issues such as process scheduling, message-passing, synchronization and data management.

With the advent of parallel computing, automatic detection of parallelism has become a topic for active research in the scientific community. The large number of sequential programs, which take up a lot of computational time on single processor machines, need to be ported on to clusters of workstations to harness its computational power and obtain significant reductions in the running times of programs. One way to achieve this is to manually rewrite the sequential programs to fit into the message-passing paradigm, however this is time-consuming, tedious and error-prone. Hence to automate the process, compiler technology is needed to compile sequential programs to distributed machines. This thesis provides the framework for building a robust and efficient parallelizing compiler which analyzes programs at compile-time with a view to reduce the computational times as much as possible. FORTRAN 77 and C are two languages in which the majority of large sequential programs are written, hence our parallelizing compiler has been developed to work on them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel computing represents the only plausible way to continue to increase the computational power available to scientists and engineers today [14]. However, until parallel computers become easy to program and use, they are not likely to be widely successful. Not so long ago, the problem with using parallel machines used to be that sequential programs had to be rewritten in a programming language that explicitly reflected the architecture of the underlying machine. The reason for this is that parallel machines are more intricate and demand a deeper understanding from those users who have to exploit machine features through specific language constructs.

MIMD distributed-memory machines such as the Intel Paragon provide the most difficult programming model [14]. Users are compelled to write message-passing FORTRAN 77 programs that deal with separate address spaces, synchronizing processors, and communicating data using messages. This is a time-consuming process and is also very tedious and prone to errors. As a side effect, significant blow ups in source code size result.

Parallel programs are extremely machine-specific. As a result, scientists are discouraged from utilizing parallel machines because there is a chance that all the effort spent into learning the intricacies of the underlying machine might go in vain whenever the program changes or a new architecture arrives for this parallel machine. This problem can be solved by developing the compiler technology needed to establish a programming model which is machine-independent. Such a model must be easy-to-use, but at the same time, perform with an acceptable degree of efficiency on different parallel architectures, atleast for data-parallel scientific codes. This thesis is directed towards building a robust and efficient parallelizing compiler for FORTRAN 77/C

programs normally in use in the industrial and scientific environment.

An emerging trend has been to use n processors in a computing system in the hope that an n-fold speedup can be achieved. Parallel computer designs have evolved through ILLIAC IV of the early 1970s to the state of the art "massively parallel" computers of today.

The **M**assively **P**arallel **P**rocessors (MPPs) and distributed computing are the two major developments which have given a tremendous boost to scientific computing. MPPs are machines built as a large collection of workstation-class [1] nodes connected by a dedicated, low-latency, high-speed network. They offer enormous computational power and are used to solve grand challenge problems such as global climate modeling, genetic engineering, particle physics, digital signal processing, finite element analysis, etc.

Distributed computing is a process whereby a set of computers connected by a **C**luster **O**f **W**orkstations (COW) (also called **N**etwork **O**f **W**orkstations (NOW), we shall be using this term throughout this thesis) are used collectively to solve a single large problem. Because of recent technology advances, NOWs are poised to become the primary computing infrastructure for science and engineering, from low-end interactive computing to demanding sequential and parallel applications [3]. A NOW provides a *global single system view* which means that a single parallel program runs on a large collection of nodes as a single entity, instead of an arbitrary collection of processes. A NOW can be thought of as a *virtual machine*, a cluster of powerful desktop machines connected by a high-speed network, that appears logically to the user as a single large parallel computer. Files can be uniformly accessed across all the nodes. In addition, the processes are scheduled as a single unit, so the constituents of a parallel program actually run in parallel.

1.1 Why NOW now?

The concept of utilizing the resources over a network has been around nearly since the evolution of computer networking, and parallel computing on networks of workstations is fast emerging as the latest trend. Some reasons why NOWs are becoming more and more attractive are [3]:

[1] The term workstation is used to refer generically to the computer system designed for the desktop.

- *Fast network speeds* : Switched local area networks allow bandwidth to scale with the number of processors and low-overhead communication protocols have made possible very fast communication over a LAN.

- *Powerful workstations* : Workstations now have enormous computational power. In addition to processor performance, a typical workstation offers huge amount of memory and disk capacity.

- *The I/O bottleneck* : Though processors are getting faster, disks are improving mostly in capacity but their performance in terms of disk access times remains almost the same. Hence, further increases in processor performance will yield little improvement for the end-user, since the processor will be idling away more and more time waiting for I/O operations. NOWs offer a better alternative. A huge pool of memory potentially exists on the network (referred to as network RAM); hence this is bound to reduce the number of disk accesses and improve performance. Also, this memory can be accessed far more quickly than local disk. In addition, file system performance can be improved by cooperative file caching, i.e., cooperatively managing the file caches on each client workstation [3].

Hence, NOWs can support high-performance parallel applications of today, which require processors with high floating-point performance, networks with bandwidth that scales with the number of processors, parallel file I/O and low-overhead communication.

1.2 Motivation

Networks of workstations are becoming more and more common, and their performance is continually increasing. They have finally begun to move from university and national laboratories in the academic and scientific community into the mainstream of commercial computing. This move has been caused by the advances in technology, spiraling costs and an ever-increasing need for more computing power. The workstations that are interconnected by high-speed networks are being used to solve a large number of computationally-intensive applications. ATM technology has made possible dramatic improvements in network speeds, of the order of 1 Gbps. These

advances in high-speed networking promise high throughput and make it possible to utilize distributed computing for many many years to come.

1.3 Problem Statement

With the advent of parallel computing, automatic detection of parallelism has become a topic of burgeoning interest to the scientific community. There exist a whole lot of huge sequential programs which take up a lot of computational time and effort. Hence, it is very desirable to run such programs on a NOW to harness its computational power and obtain significant improvements in the running times of these programs. Current NOW systems are based on the message-passing model of parallel computing. Programmers are required to explicitly deal with issues such as process scheduling, message-passing, synchronization and data management. These are additional overheads which have been introduced because of the programming paradigm and are not part of the problem the user is solving. There is also a shift from the users' view of a single, monolithic, local memory of a sequential machine to a distributed view of memory across multiple nodes. This makes distributed computing difficult. Moreover, message passing depends on the organization of the data and a change in the data organization makes it necessary to rewrite the whole program.

In order to make programming for distributed environments easier, the low-level details like message-passing among machines, synchronization among tasks and data management should be hidden from the user and handled by the compiler. The problem is to design a compiler for data parallel scientific languages which can handle these issues. The issues handled by our parallelizing compiler are :

- Generation of message-passing code
- Synchronizing among the tasks running on different processors using non-blocking sends and blocking receives for communicating non-local data among processors.
- Handling array data distributions as specified by the programmer.

1.4 Related Work

Existing parallelizing compiler systems can be broadly classified into two categories, depending on the array references they handle. In the first category are compilers

which handle only regular array references, such as SUPERB, ID NOUVEAU, FORTRAN D. In the second category are compilers which also handle irregular array references, like KALI. We discuss below in brief some of these systems :

1. **SUPERB** [28] is a semi-automatic parallelization tool that supports arbitrary user-specified contiguous block decompositions. It translates FORTRAN 77 programs into message-passing FORTRAN for the Intel iPSC, the GENESIS machine, and SUPERNUM. The user specifies the distribution of arrays through an interactive language. Program analysis and dependence analysis information, using both intra and inter procedural analysis, is computed and made available to the user, who may select individual transformations or request services via menus. The data dependence information is used to apply loop distribution and vectorize these statements, resulting in vectorized messages. SUPERB does not perform run-time analysis, nor does it allow redistribution of arrays.

2. **ID NOUVEAU** [22] is a functional language enhanced with block and cyclic distributions. Initially, send and receive statements are inserted to communicate each non-local array access. Message vectorization is applied to combine messages for previously written array elements. Loop fusion and strip mining are used when writing array elements. It differs from other systems in that it produces distinct programs for each node processor and hence takes more time for compiling.

3. **KALI** [15] is an imperative language for MIMD distributed memory machines. The compiler for KALI translates KALI code for execution on the Intel iPSC/2. A unique feature of this compiler is that it was the first to support both regular and irregular computations. However, it does not provide dependence analysis, so the programmer must declare all parallel loops. Instead of deriving a parallel program from the data decomposition, KALI requires that the programmer explicitly partition loop iterations onto processors using an 'on clause'.

4. **CRYSTAL** [16] is a high-level functional language compiled to distributed memory machines using both automatic data decomposition and communication generation. Program analysis and optimization are different because it targets a purely functional language.

5. **FORTRAN D** [14] is a version of FORTRAN extended with data decomposition specifications. It performs data dependence analysis so as to exploit

parallelism from sequential constructs without relying on single-assignment semantics (e.g., CRYSTAL, ID NOUVEAU) or explicitly parallel programs (e.g. KALI). It supports irregular as well as dynamic distribution.

1.5 System overview

The compiler developed takes a sequential FORTRAN 77/C program as input and generates a C program with MPI calls inserted at appropriate points in the program. Message passing and scheduling of SPMD processes on a NOW is being done using MPI.

The data parallel environment is shown in Figure 1.1. It comprises of the parallelizing and MPICC compilers, a portable MPI message-passing library, and the NOW.

1.6 Organization of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 describes the programming environment and tools used in the thesis.
- Chapter 3 discusses in detail the different types of analyses done by our parallelizing compiler.
- In Chapter 4, we present the detailed implementation of the techniques discussed in Chapter 4.
- In Chapter 5, we show the results obtained after running the parallel programs generated by our compiler on the NOW.
- Chapter 6 presents the conclusions and throws some light on future direction.

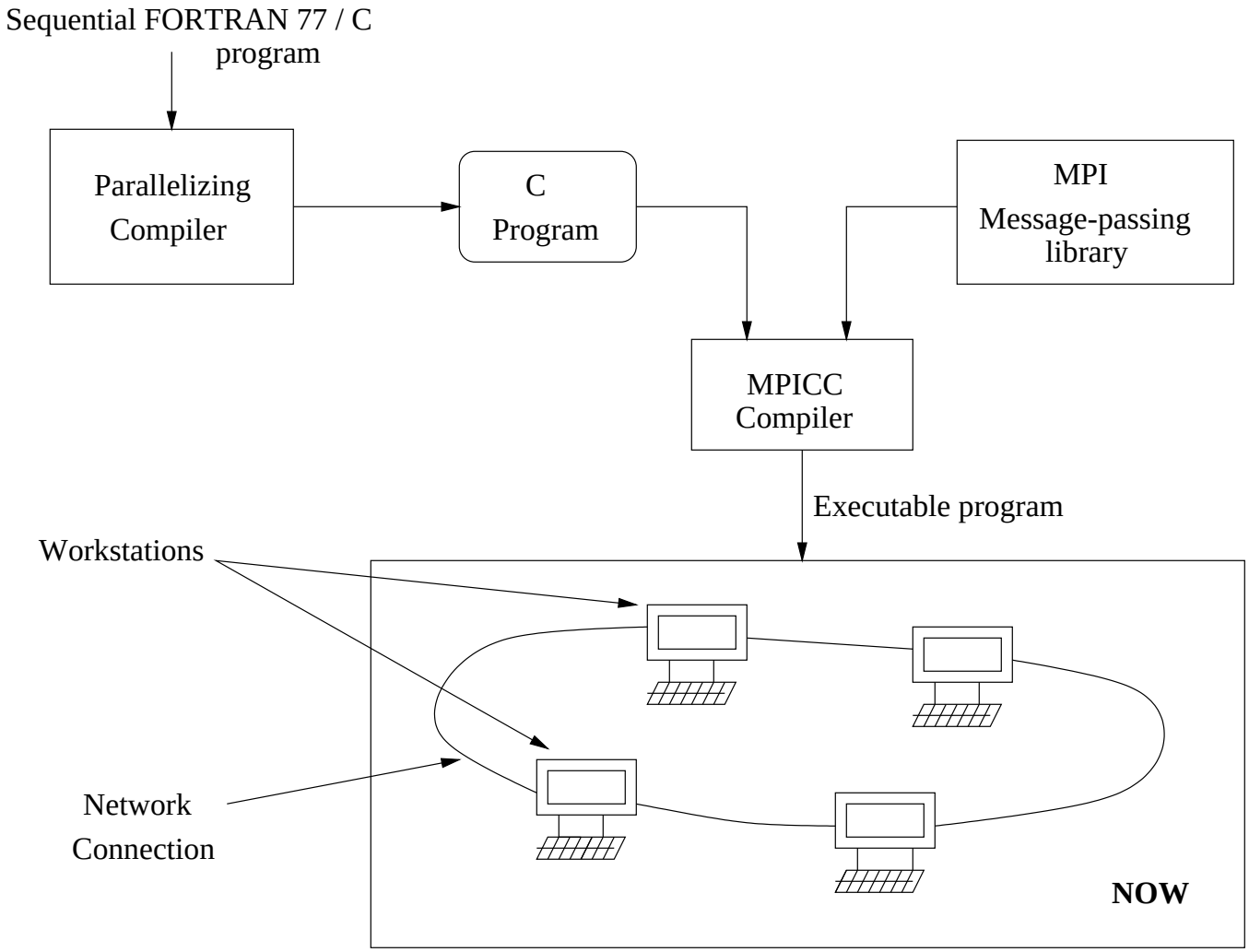


Figure 1.1: Data Parallel Environment

Chapter 2

The Programming Environment

In this chapter, we shall be describing the programming environment we used for building the compiler. Our compiler translates sequential FORTRAN 77/C programs into **S**ingle-**P**rogram, **M**ultiple-**D**ata (SPMD) form with explicit message-passing that execute directly on the nodes of the distributed-memory machine. We make use of the SUIF (**S**tanford **U**niversity **I**ntermediate **F**ormat) compiler framework developed at Stanford University. The compiler back end, which generates the output C program annotated with MPI calls, is implemented with the aid of a tool provided with SUIF called SUIF Builder. MPICH, a portable implementation of the **M**essage-**P**assing **I**nterface (MPI), provides the message-passing library. We start by giving a brief overview of the SUIF system, followed by a discussion about the Builder tool. Then we briefly mention the SUIF data dependence library. Next, we give an overview of message-passing and why MPI is accepted as the standard in message-passing systems today. We finally end by giving brief details about MPICH.

2.1 The SUIF library - An Overview

The SUIF compiler system is a flexible framework for advanced compiler research [25]. The primary design goal for this compiler is that it be easy to use and build upon, without sacrificing its ability to compile real-world programs. Compared to other publicly available compilers, SUIF may be somewhat inefficient and less robust, but it is much easier to modify and extend. This makes it an ideal platform for evaluating new compiler techniques.

The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. The passes are implemented as separate programs that link with the kernel contained in the SUIF library. Each pass typically performs a single analysis or transformation and then writes the results out to a file. This is inefficient but flexible. SUIF files always use the same output format so that passes can be reordered simply by running the programs in a different order. New passes can be freely inserted at any point in the compilation.

The SUIF library provides an object-oriented implementation of the SUIF intermediate format. It is written in C++. The library defines classes to represent the various elements of the intermediate format and to perform some common operations on them. It also contains the code to read and write the binary files that hold the SUIF code between passes of the compiler.

2.2 The SUIF Builder

The SUIF Builder is a tool which provides an efficient method to create or modify code within the SUIF compiler [25]. Each pass of the SUIF compiler represents an input program using a set of complex data structures. Thus, when a compiler pass needs to modify the original program, it needs to manipulate these data structures. A simple five line source program may need a few hundred complex structures to represent the program in a SUIF compiler pass. Changing a single line of the original program while executing the compiler pass means updating many of these data structures which is clearly undesirable.

The goal of the Builder is to abstract these complexities away from a programmer writing a pass for the SUIF compiler. When a programmer needs to change or add new code to the input program, the Builder provides a simple and nice abstraction. Advantages of using the Builder are [25] :

- Very simple abstraction: Creating new code and changing existing code is done using a C-like syntax. There is no need to understand the SUIF internals and the rules governing the updates to the SUIF structures.
- A powerful abstraction: The Builder has the same power as the C language in transforming an algorithm to a code segment.

- Simple and elegant code: Since many of the details are taken care of by the Builder, the code needed to do manipulations is very simple. For example, an unrolling of a loop can be performed using six lines of Builder code instead of a few pages of C++ code to explicitly manipulate the SUIF structures.

- Ease of debugging: The Builder provides extensive syntactic and semantic analysis of the code that is created.

- Maintainable code: Future changes to SUIF may be transparent to users of the Builder since many of the rules governing the SUIF system are abstracted away by the Builder.

- Single standard: When the Builder is used, all the code transformations across different passes of the compiler (at least at the loop-level) will have a single code transformation standard. Since there are thousands of different ways to do code transformations, Builder can establish some standard across passes by different programmers.

The following material discusses in brief the inner details of the Builder tool. For more details, see [25].

2.2.1 Building Blocks

The Builder uses the concept of building blocks to construct code segments. Each unit is a block (which is a C++ class), and composition of multiple units will yield another block. Thus a complex code segment is represented by a block which is a composition of smaller blocks. After constructing the blocks, the SUIF code is obtained by invoking a function to generate the SUIF structures.

2.2.2 Types

The SUIF compiler supports a full type system. Thus, when modifying SUIF code, it may be necessary to create or change the types. The various types provided are the base types (void, char, short, int, int16, int32, int64, long, unsigned, unsigned16, unsigned32, unsigned64, float, float64 and double), pointer types, array types, function types, structure and union types, etc.

2.2.3 Creating expressions and statements

The Builder library provides a rich set of functionality to use most of the arithmetic and logic operators in creating expressions. Expressions are built-up using other expressions, variables and constant values.

2.2.4 Creating Code with Control Flow

Many structured and unstructured control flow constructs such as FOR, WHILE and DO loops, IF conditionals and RETURN and GOTO instructions can be used within the Builder.

2.2.5 Support for Debugging

The Builder system was created to make life easier for compiler writers. Most of the Builder design decisions were taken in favor of ease of programming over efficiency. Proving good debugging support was an integral part of this philosophy. Debugging support was provided at two levels: compile-time support obtained from the C++ compiler and the run-time support programmed into the Builder system.

2.2.6 Error Checking

- *Compile-time Error Checking*

When compiling the compiler-pass, a limited amount of error checking of the Builder structures is provided by the C++ compiler. If operator overloading is used, C++ compiler checks for any syntax errors. Since all the Builder structures are of the same block class, no semantic checking is provided at compile time.

- *Run-Time Error Checking*

When running the compiler pass, the Builder library is invoked. All semantic and syntax checks are done at this level. When an error is found, the Builder prints an error message and tries to continue executing. This is done to help debugging since the first instance of finding an error may not be the exact location of the error.

- *Syntax and Semantic checking*

A nearly complete semantic and syntax check is implemented in the Builder system. If a semantically or syntactically incorrect Builder structure is found, an error message is printed with the line information of the Builder library where the error was located.

In addition, the Builder checks the code that is generated for possible errors and warns the user.

2.3 The SUIF data dependence library

Data dependence analysis involves analyzing the dynamic memory reference behavior of array operations so that compilers will only parallelize loops in the cases where any resultant reordering of memory references does not change the sequential semantics of the program. It is a powerful analysis which facilitates parallelization of loops containing array accesses as well as various forms of loop transformations such as loop tiling, strip mining, etc. In general, data dependence analysis is undecidable, and compilers must conservatively approximate array reference behavior, thus sequentializing parallel loops. Therefore, traditional data dependence analysis research has concentrated on the simpler problem of affine [1] memory disambiguation. In this simpler domain, array references and loop bounds are assumed to be linear integer functions of loop variables. Many algorithms have been developed that conservatively approximate even this simpler problem. By using a series of algorithms, each one guaranteed to be exact for a certain class of input, the SUIF compiler, in practice, solves exactly and efficiently the affine memory disambiguation problem [17].

2.4 An overview of Message Passing

Message-passing is a programming paradigm used widely on parallel computers, especially Scalable Parallel Computers (SPCs) with distributed memory, and on NOWs [24]. Conceptually, the idea behind message-passing is simple - multiple processors of a parallel computer run the same or different programs, each with private data. Data is exchanged between processors as and when needed. A message is

[1] Affine subscript expressions are subscript expressions that are affine functions of the loop induction variables [27]. They are also called linear subscripts, since the expressions are linear combinations of the induction variables.

transmitted by a sender processor to a receiver processor. A processor can either be a sender or a receiver at any time. The sender processor can either wait for an acknowledgement after sending or it can continue execution. The receiver processor checks a message buffer to retrieve a message. If no message is present, the processor can continue execution and try again later or wait until a message is received. Multiple sends and receives can occur simultaneously in a parallel computer. All processors are interconnected by a network and can exchange messages with each other. The routing of messages is handled by the underlying operating system.

Message-passing has been used successfully to implement many parallel applications. The advantages of a message-passing model are ease-of-use and portability. A message-passing program written using a standard communication protocol can be ported from a network of PCs (or a network of heterogenous machines) to a massively parallel supercomputer without major changes. However, the disadvantage is the added programming required. Adding message-passing code to a large program requires considerable time. This is where a compiler for message passing comes into the picture and simplifies the programmer's job. It consists of the base compiler and is equipped with a communication library.

2.4.1 MPI - A Message Passing Interface

MPI is a standard and portable message-passing system designed to function on a wide variety of parallel computers [18]. The standard defines the syntax and semantics of a core of library routines for writing portable message-passing programs in FORTRAN, C and C++.

Why MPI ?

MPI combines the most attractive features of a number of existing message-passing systems, instead of selecting one of them and making it the standard. Thus, MPI has been strongly influenced by Intel's NX/2 [21], Express [20], nCUBE's Vertex [19] and PARMACS [13] and in systems sold by IBM, Intel, Meiko, Cray Research and nCUBE. Other important contributions have come from Zipcode [23], Chimp [23], PVM [7], Chameleon [10] and PICL [8].

MPI was first proposed to be an intermediate level message passing standard by Dongarra *et al* [5]. They argued that it is possible to define a message-passing

standard at a number of levels. The lowest level, closest to the hardware, could consist of routines for moving packets along wires. Above this channel-addressed level could be a process-addressed level (with more than one process on each processor), such as that defined by NX or Vertex on the iPSC and nCUBE machines, the Express system, or the PARMACS message-passing macros. Higher-level abstractions such as Linda [9] would lie below this level. Each level could be built using the levels below it and these successive levels form a series of levels, which Dongarra *et al* [5] refer to as the "Onion Skin Model of the distributed communication environment", since they are analogous to the multiple skins of an onion, with the hardware being at the center. Now, different people favour different levels. For example, a naive user would prefer high-level abstractions, such as virtual shared memory, so that the message-passing details are hidden. On the contrary, compiler writers would like to create a fast parallel compiler, and so would prefer a low-level standard for making use of small, fast messages. Finally, an expert application developer might be prepared to forsake some ease-of-use for additional speed, and so would like an intermediate-level standard that provides a set of efficient primitives for point-to-point message-passing. This is where MPI is useful.

The idea of an intermediate level standard is also helped by the fact that the hardware of various distributed-memory computing systems are quite different and so it is difficult to impose a low-level standard that is efficient on all machines. Furthermore, it is possible to define higher-level standards on top of this intermediate level, thus making it open and extendable. For more information about how MPI combines the features of various other message-passing systems, refer [5].

The advantages of using MPI for parallel programming are :

- It has become the leading standard for message-passing libraries and is portable across different machines. This means that the same message-passing code can be executed on a variety of machines as long as the MPI library is available, while some tuning might be needed to take best advantage of the features of each system. Since efficient MPI implementations exist across a wide variety of computers, a high degree of flexibility is provided in code development, debugging and in choosing a platform for production runs.

- Multiple implementations of MPI are freely available for all parallel computers, including heterogenous NOWs and single parallel machines.

- MPI allows or supports scalability through several of its design features. For example, an application can create subgroups of processes, that in turn, allows collective communication operations to limit their scope to the processes involved.

2.4.2 MPICH - A portable MPI Implementation

MPICH [4] is a freely-available, portable implementation of MPI developed at the Mathematics and Computer Division of the Argonne National Laboratory. It contains, along with the MPI library itself, a programming environment for working with MPI programs. The programming environment includes a portable startup mechanism, several profiling libraries for studying the performance of MPI programs, and an X-interface to all the tools. MPICH has a rich set of point-to-point communication routines as well as a large set of collective communication routines for communication among groups of processes.

Chapter 3

The parallelizing compiler

This chapter discusses the various kinds of analyses done by our parallelizing compiler on the source program. A source-to-source translation is performed on the input C/FORTRAN 77 program resulting in a program annotated with calls to the message-passing library. The compiler assumes that the program will be executed on a distributed memory machine (NOW) according to the SPMD programming model. This model requires that each participating processor execute the same program; parallelism is obtained by applying the computation to different parts of the data domain simultaneously. We start by discussing the basic model for compiling to distributed memory machines.

3.1 Basic Model

There are three domains that are manipulated while compiling to distributed-memory machines :

- Iteration space, $\mathcal{I}$
- Array elements space, $\mathcal{A}$
- Processor space, $\mathcal{P}$

The occurrences of variable names in a statement S are classified as definitions and uses, depending on whether they are assigned to, or their value is read. The corresponding sets for a statement S are denoted by DEF(S) and USE(S). For example, in the following statement:

S: A(i) = B(i+1) + A(i-1)

the corresponding sets are : DEF(S) = {A(i)} and USE(S) = {B(i+1), A(i-1)}.

The following theory is discussed in detail in [2]. Each array access function in the source program specifies the data used by each iteration in the loop. That is, each read or write access function, denoted by f_r, f_w respectively, maps an iteration to the array indices of the data read or written, i.e., f_r, $f_w : \mathcal{I} \longrightarrow \mathcal{A}$. The user-specified data decomposition maps each array location to a processor, i.e., $\mathcal{D} : \mathcal{A} \longrightarrow \mathcal{P}$. Finally, from the access function f and data decomposition $\mathcal{D}$, the compiler automatically derives the computation decomposition $\mathcal{C}$ which maps each iteration in the loop to a processor, i.e., $\mathcal{C} : \mathcal{I} \longrightarrow \mathcal{P}$

To derive the computation decomposition, the compiler applies the *owner computes rule* : each assignment statement is performed by the processor that owns the data. Therefore, given a write access function f_w and a data decomposition $\mathcal{D}$, the computation decomposition is $\mathcal{C} = \mathcal{D}f_w$. Under the owner computes rule, no communication is needed to implement the write access. Communication is needed for a read access in iteration i iff the data read is resident on a different processor, i.e., $\mathcal{C}i \neq \mathcal{D}f_r i$. In the above example, no communication is needed for the reference R $\in$ DEF(S), whereas communication is needed for a R $\in$ USE(S) iff R resides on a different processor.

The relationships between iteration space, array space and the processor space are shown in the Figure 3.1 [2]; processor p_r receives data from processor p_s iff $p_r \neq p_s$. a_w and a_r represent write and read instances of array references used in the program respectively. Hence, f_w maps from iteration i to a_w and f_r maps from iteration i to a_r. The computation decomposition $\mathcal{C}$ maps from iteration i to a processor p_r due to the *owner computes rule*.

3.2 Structure of the Compiler

The principal components of the parallelizing compiler which performs a source-to-source transformation are shown in Figure 3.2. The input language is FORTRAN 77/C, the output language is C (annotated with MPI calls) which can be directly compiled and executed by the *mpicc* compiler on the NOW.

The system consists of two main modules : the front end and the back end. The

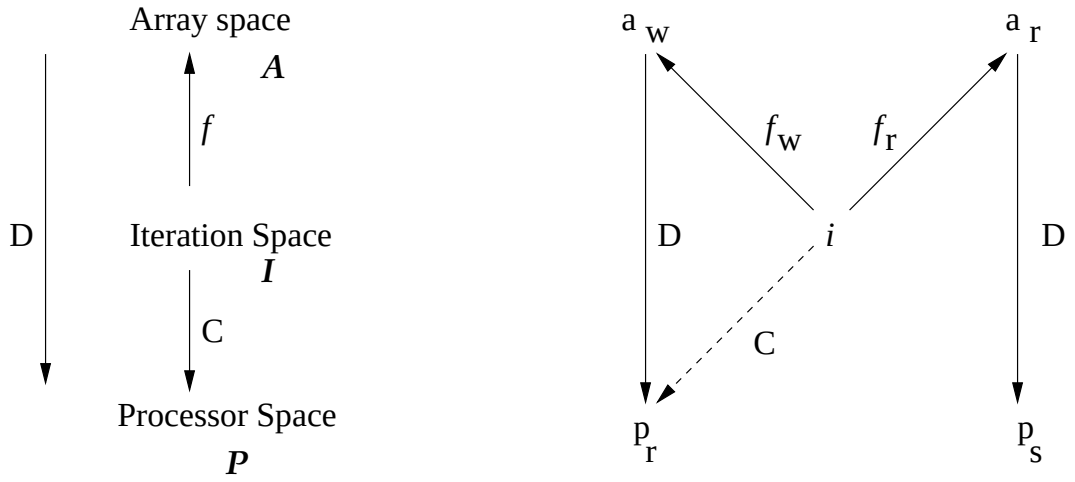


Figure 3.1: Relationship between iteration space, array space and processor space

front end performs the analysis tasks of a conventional compiler, such as lexical analysis, parsing and semantic analysis and creates an intermediate representation of the program in SUIF. Our compiler then performs scalar data flow analysis, array dependence analysis, symbolic analysis, partitioning analysis and communication analysis on this intermediate representation. The task of the back end is to generate the parallel C program from the SUIF representation, by applying parallelization transformations such as program partitioning. The code generation is performed by a tool called SUIF Builder provided with the SUIF compiler framework.

3.3 Data decomposition specifications

The mapping of computation onto the processors of a parallel machine is termed the *computation decomposition* of the program. Similarly, the placement of data into the processors' local memories is called the *data decomposition*. Selecting an optimal data decomposition for a given program is one of the toughest intellectual steps in developing data-parallel scientific codes. Though many techniques have been developed for automatic data decomposition, the compiler is not able to choose an efficient data decomposition for all programs. So, like most systems (High Performance FORTRAN, FORTRAN-D [14], Vienna FORTRAN), our compiler solicits the programmer's help in supplying the data decompositions. These decomposition specifications represent how arrays should be distributed onto the actual parallel machine. This is called the machine mapping caused by translating the problem onto the finite resources of

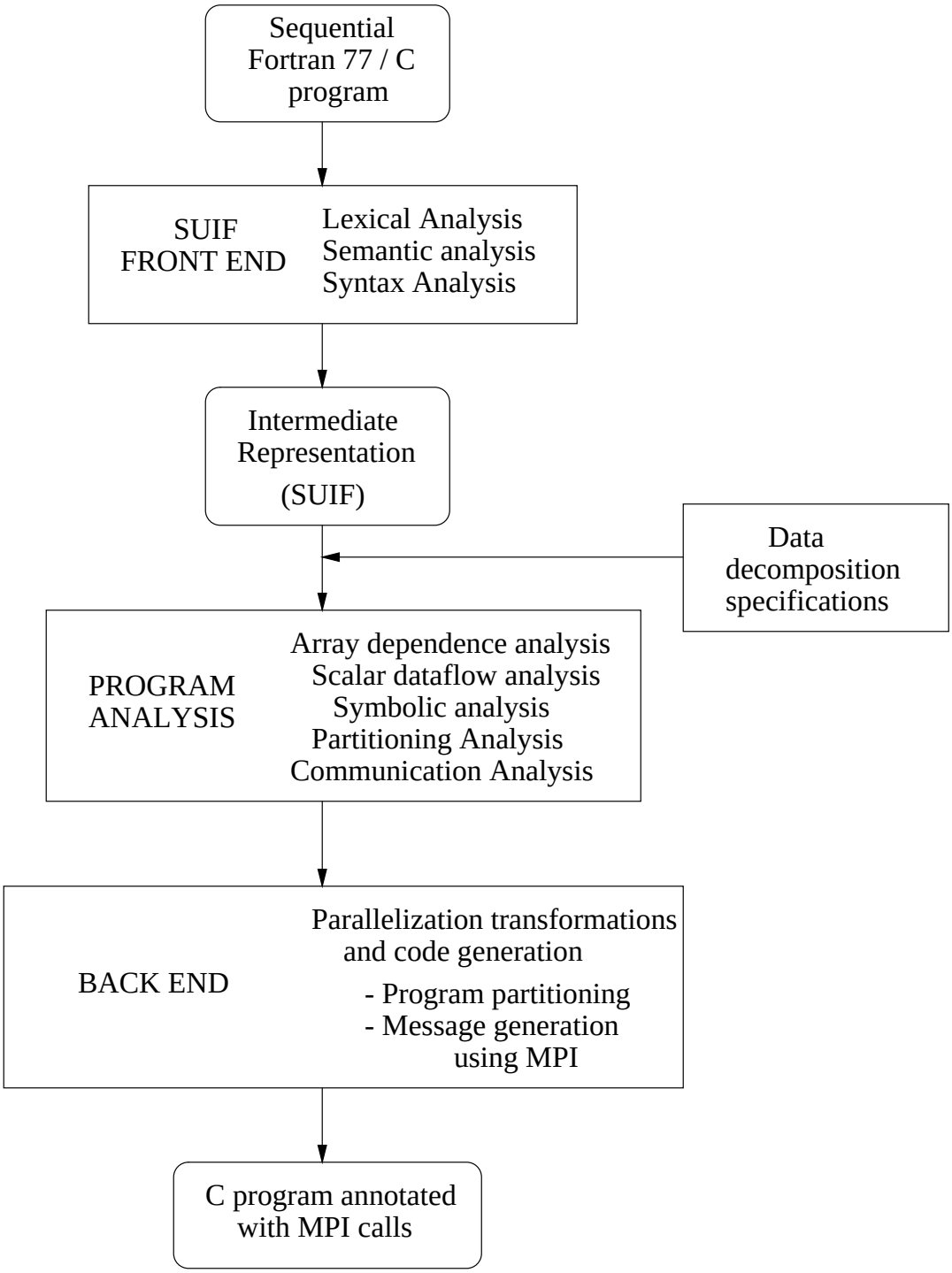


Figure 3.2: Overall structure of the parallelizing compiler

the machine. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

We use a DISTRIBUTE statement (a hybrid of the DECOMPOSITION and DISTRIBUTE statements in FORTRAN D [14]) to specify the mapping of the arrays onto the various machines of the cluster. The distribution selected affects the ability of the compiler to to minimize communication and load imbalance for the resulting program. The factors responsible for selecting a good decomposition are physical machine characteristics such as the number of processors, amount of memory per processor and communication costs between processors as well as program characteristics such as the size of distributed arrays and the computation structure.

In addition, data parallelism may either be regular or irregular. Regular parallelism can be fully exploited through relatively simple data distributions, whereas irregular parallelism require irregular data distributions and run-time processing. Our compiler accepts regular decompositions only.

Regular decompositions

Distributions are specified by assigning an independent *attribute* to each dimension of a decomposition. Two predefined attributes for regular decompositions are BLOCK and CYCLIC. The symbol "*" marks dimensions that are not distributed. Scalars are replicated, i.e., owned by all processors.

Thus, the statement "DISTRIBUTE A (*, BLOCK)" results in a column partition of the array A among the processors. Similarly, the statement "DISTRIBUTE B (CYCLIC(5), *, *) means that the rows of the array B which has three dimensions are to be partitioned in round-robin fashion with a block size of 5. If no decomposition is specified for an array, we assume that it is to be partitioned block wise in the first dimension. Let P be the total number of processors and N be the array bound. If P divides N evenly, the resulting distribution will be slightly unbalanced. The distributions can be defined as follows :

- BLOCK divides the decomposition into contiguous chunks of size $\lceil N/P \rceil$, assigning exactly one block to each processor.
- CYCLIC specifies a round-robin division of the decomposition, assigning every P^{th} element to the same processor.

- REPLICATION, denoted by *, means that every processor has a copy of the element.

The distribution specified determines the communication patterns as well as the load balance among the processors. A cyclic distribution often tends to improve load balance when computation is centered along a small part of the array; however it tends to increase communication, as adjoining elements will be on different processors. On the other hand, block distributions tend to have opposite characteristics.

Using the above definitions, the distribution function which maps an array subscript to the appropriate processor for various patterns can be shown as in Table 3.1.

BLOCK	D(i) = {i * P / N}
CYCLIC(k)	D(i) = {$\lfloor$i/k$\rfloor$ mod P}
REPLICATION	D(i) = {p $\mid$ 1 $\leq$ p $\leq$ P}

Table 3.1: Distribution function

Primitives used

Before we go into the details of the various analyses performed, we shall define the following primitives to be used throughout this chapter :

- Owner Primitives
  1. **PROC_ID**() : It returns a unique integer representing the local processor.
  2. **OWNER**(R) : It returns the unique processor which owns the array reference R.
- Message primitives
  1. **SEND**(R, P_r) : Local processor (PROC_ID()) sends the array reference R to processor P_r. It is non-blocking, i.e., it does not wait for the matching receive to complete.
  2. **RECV**(R, P_s) : Local processor (PROC_ID()) does the receive for the array reference R from the processor P_s. It is blocking, i.e., it will block till the matching send is completed.

3.4 Run-time resolution

Run-time resolution is a simple and straightforward technique to generate code that explicitly calculates the ownership and communication sets for each array reference at run-time [22]. Every processor examines each statement in the program and complies with the following rules of thumb :

- If the processor owns an array element, it is responsible for computing its defining expression and recording its value.
- The processor that owns the array element must communicate its value to any other processor needing this value.
- Conditionals, function calls and loops are executed by all processors.

For example, application of run-time resolution to the FORTRAN program shown in Figure 3.3 results in the code shown in Figure 3.4.

```
      PROGRAM P
      REAL A(100)
      DO 100 I = 1, 55
        A(I) = A(I+10) + 2
100   CONTINUE
      END
```

Figure 3.3: Sample FORTRAN program

Run-time resolution does not require much compiler analysis. However, the resulting programs execute much slower than the original sequential code, so they are extremely inefficient. The performance of run-time resolution is poor since parallelism is mostly lost because every processor must execute the entire program. Even worse, not only does the program explicitly have to check every variable reference, it also generates a message for each nonlocal access. Hence if compile-time analysis is not performed, the compiler is forced to rely on run-time techniques.

```
      PROGRAM P
      REAL A(100)
      DO 100 I = 1, 55
        IF (PROC_ID() .EQ. OWNER(A(I+10)) THEN
          SEND A(I+10) to OWNER(A(I)
        ENDIF
        IF (PROC_ID() .EQ. OWNER(A(I)) THEN
          RECV A(I+10) from OWNER(A(I+10)
          A(I) = A(I+10) + 2
        ENDIF
  100 CONTINUE
      END
```

Figure 3.4: Program after applying run-time resolution

3.5 Program Analysis

Now we shall describe in detail the various kinds of analyses done on the source program at compile time.

3.5.1 Dependence Analysis

Dependence analysis is the compile-time analysis of control flow and memory accesses to determine a statement execution order that preserves the meaning of the original program. A *data dependence* between two array references R_1 and R_2 indicates that they read or write a common memory location in a way that requires their execution order to be maintained. R_1 is called the *source* and R_2 is called the *sink* of the dependence if R_1 must be executed before R_2. If R_1 is a write and R_2 is a read, we call the result a *flow* dependence. There are four different types of dependences [27] :

- *Flow* dependence occurs when a variable is assigned or defined in one statement and used in a subsequently executed statement.
- *Anti*-dependence occurs when a variable is used in one statement and reassigned in a subsequently executed statement.
- *Output* dependence occurs when a variable is assigned in one statement and reassigned in a subsequently executed statement.

- *Input* dependence occurs when the same value of a variable is read in two particular statements.

Anti-dependence and output dependence arise from the reuse or reassignment of variables, and are sometimes called false dependences. Flow dependence is also called *true* dependence since it is inherent in the computation and cannot be eliminated by renaming variables.

Dependences may be either loop-independent or loop-carried. *Loop-independent* dependences occur on the same loop iteration; *loop-carried* dependences occur on different iterations of a particular loop. The *level* of a loop-carried dependence is the depth of the loop carrying the dependence. Loop-independent dependences have infinite depth. The number of loop iterations separating the source and the sink of the loop-carried dependence may be characterized by a dependence *distance* or *direction*.

3.5.2 Scalar dataflow Analysis

Any scalar assigned in a loop will always generate one or more loop-carried data dependence relations. Such scalar assignment statements present a special challenge for our compiler. The compiler has the option of trying to remove them, synchronizing to satisfy them, or using sequential code.

Naive application of the owner computes rule would cause every processor to execute the assignment on all iterations. However, often the assignment can be partitioned because its value is used only in the current loop iteration. All such analysis is done by the scalar dataflow analysis phase. Control flow, control dependence and live range information are computed. Scalar and array variables are declared "private" with respect to a loop if their values are used only within the current loop iteration; this is useful for eliminating unnecessary computation and communication when the loop iterations are divided amongst the processors.

3.5.3 Symbolic Analysis

Constant propagation, simple induction variable elimination, common subexpression elimination, loop invariant expression recognition are performed during the symbolic analysis phase of the parallelizing compiler. The goal is to provide a simplified program representation that improves program analysis and optimization. Forward

propagation is also performed; the calculation of local variables are propagated into the uses of those variables whenever possible. The idea is to give more information about loop bounds and array indexing for doing dependence analysis and loop transformations. In addition, the compiler tries to construct FOR constructs out of loop constructs such as DO or WHILE (for C programs) whenever a suitable index variable and bounds can be found. This is helpful in extracting more parallelism.

3.5.4 Partitioning Analysis

The program partitioning analysis phase of our compiler divides the overall data and computation among processors. This is accomplished by first partitioning all arrays onto processors (specified by the user using decomposition specifications discussed in Section 3.3) and then using the *owner computes rule* to derive the functional decomposition of the program.

- *Some definitions*

Iteration and index sets, RSDs

An *iteration set* is simply a set of loop iterations - it describes a section of the work space. An *index set* is a set of locations in an array - it describes a section of the data space. The compiler constructs iteration or index sets using *regular section descriptors* (RSDs) as in the FORTRAN D compiler [14]. RSDs are widely used in our compiler as an internal representation. They are compact representations of rectangular or right-triangular array sections and were originally created to summarize array side effects across procedure boundaries [12]. The union and intersection of RSDs can be calculated inexpensively, making them highly useful for our compiler.

An RSD can be denoted as $[l_i{:}u_i{:}s_i, \ ...]$, where l_i, u_i and s_i indicate the lower bound, upper bound and step of the ith dimension of the RSD respectively. If not explicitly stated, a default unit step is assumed. In loop nests or multidimensional arrays, the leftmost dimension of the RSD corresponds to the outermost loop or the leftmost array dimension. The other dimensions are listed in order.

Local index sets

As the first step in partitioning analysis, the local index set for each array is computed. The local index set is the local array section owned by every processor. This creates the data partition used in the program.

The analysis required can be illustrated using the Jacobi code shown in Figure 3.5.

```
REAL A(100, 100), B(100, 100)
INTEGER I, J, K, TIME
TIME = 50
DO K = 1, TIME
  DO J = 2, 99
    DO I = 2, 99
S1:    A(I,J) = (B(I,J-1) + B(I-1,J) + B(I+1,J) + B(I,J+1)) / 4
    ENDDO
  ENDDO
  DO J = 2, 99
    DO I = 2, 99
S2:    B(I, J) = A(I, J)
    ENDDO
  ENDDO
ENDDO
```

Figure 3.5: Jacobi code

Assuming that we are compiling for a four-processor machine and the programmer has specified the data decomposition as :

DISTRIBUTE A (*, BLOCK)

DISTRIBUTE B (*, BLOCK)

i.e. the second dimensions of both arrays are block-distributed. The local index sets for both A and B on each processor are given as :

p1 [1:100, 1:25]

p2 [1:100, 26:50]

p3 [1:100, 51:75]

p4 [1:100, 76:100]

Execution iteration sets

An execution iteration set (or simply execution set) can be defined as the set of loop iterations executed by each processor. It can be computed by taking the intersection of the appropriate local index set with the iteration set of the enclosing loops in the correct order of the indices of the array reference. The algorithm is shown in Figure 3.6.

Algorithm for computation of execution iteration set
/* λ^A(p) represents the local index set of the array A on processor p */

for each array **A do**
  for each processor $p \in$ P **do**
    Compute $\lambda^A(p)$ from array **A**'s distribution, D
    Execution iteration set = λ^A(p) $\bigcap$ { iteration set of enclosing loops in order}
  endfor
endfor

Figure 3.6: Algorithm for computation of execution iteration set

Hence the execution iteration sets in the Jacobi code example are :

p1 [2:99, 2:25]

p2 [2:99, 26:50]

p3 [2:99, 51:75]

p4 [2:99, 76:99]

Once the execution iteration sets have been computed, the next step is to calculate the communication sets.

3.5.5 Communication Analysis

Communication analysis determines which array references cause nonlocal data accesses. For each processor P, we compute send sets to every other processor Q, where P $\neq$ Q, which represent the set of elements to be communicated among the processors due to cross-processor dependences. The computation is shown in Figure 3.7.

Using the above algorithm, the send sets for reference B(i, j-1) in statement S1 of the Jacobi code example are computed as follows :

The local index set of array B is same as that of array A. After applying the subscript functions of B(i, j-1) to the execution iteration set, we get, [2:99, 1:24]@p1,

Algorithm for computation of Send sets

for each R $\in$ DEF(S) **do**
  for each processor $q \in$ P **do**
    Compute the execution iteration set $\xi(q)$ for processor q using the previous algorithm.
  endfor
endfor
for each R $\in$ USE(S) **do**
  for each processor p, $p \in$ P **do**
    for each processor q, $q \neq p$, $q \in$ P **do**
      Calculate $f(\xi(q))$, where f is the array access function in R.
      Send_set (R, p, q) = $f(\xi(q)) \bigcap \lambda(p)$
    endfor
  endfor
endfor
for each processor p, $p \in$ P **do**
  for each processor q, $q \neq p$, $q \in$ P **do**
    Send_set (p, q) = ϕ
  endfor
endfor
for each R $\in$ USE(S) **do**
  for each processor p, $p \in$ P **do**
    for each processor q, $q \neq p$, $q \in$ P **do**
      Send_set (p, q) = Send_set (p, q) $\bigcup$ Send_set (R, p, q)
    endfor
  endfor
endfor

Figure 3.7: Algorithm for computation of send sets

[2:99, 25:49]@p2, [2:99, 50:74]@p3 and [2:99, 75:98]@p4. After taking the intersection with the local index sets of B, the send sets are :

Send-set (p1, p2) = [2:99, 25]
Send-set (p2, p3) = [2:99, 50]
Send-set (p3, p4) = [2:99, 75]

3.6 Code generation

Once the program analysis phase is complete, the code generation phase of our compiler utilizes information concerning local index sets and iteration sets and RSDs to create the actual SPMD node program with message-passing calls.

3.6.1 Program Partitioning

During partitioning analysis, our compiler applies the *owner computes rule* to calculate the execution iteration set for each statement. One of the goals for code generation is to modify the program to ensure that the iterations are divided correctly among the processors while enforcing the owner computes rule. This may be accomplished by a combination of reducing loop bounds (or *loop bounds reduction*) and guarding individual statements (or *guard introduction*). Both approaches require calculation of the execution iteration sets for statements in a loop.

- *Loop bounds reduction*

Evaluation of guard statements at run-time leads to execution overhead. Hence, our compiler first tries to reduce the loop bounds wherever possible to avoid evaluating guard expressions while at the same time, ensuring that the owner computes rule is enforced. Each processor then will only execute iterations in the unioned iteration execution sets of all statements in the loop. The compiler then inserts code to calculate boundary conditions, as shown by the bounds generated for the j loop in Figure 3.10.

A simple algorithm for performing loop bounds reduction is shown in Figure 3.8. The execution iteration sets of all the lhs array references in the loop are unioned together. The resulting set represents all the iterations on which an assignment needs to be executed by the processor. Code is inserted for assigning values to the bounds before the loop nest and the loop is rewritten by reducing the loop bounds to this reduced iteration set.

- *Guard introduction*

Loop bounds reduction eliminates the need for masks when all assignment statements in the loop have the same iteration set and so these statements always execute in the

Algorithm for performing loop bounds reduction

for each FOR loop nest $k = l$ to m by s **do**
  reduced_iter_set = ϕ
  for each lhs array reference A in loop **do**
    reduced_iter_set = reduced_iter_set $\bigcup$ execution iteration set of A
  endfor
  Generate code for changing bounds of loop nest to those in reduced_iter_set
  Replace FOR loop by another FOR loop with reduced bounds
endfor

Figure 3.8: Algorithm for performing loop bounds reduction

reduced loop. However, when loop nests contain multiple assignment statements to distributed data structures, the iteration set of each statement for a processor may differ, limiting the number of guards eliminated through loop bounds reduction. Hence, in these cases, our compiler introduces masks for the statements that are conditionally executed, thus enforcing the owner computes rule.

Hence, guard introduction replaces each statement S of the program by the associated masked statement :

if (mask(S)) { S; }

where,

- mask(S) = OWNER (A(*f*)) if S is an assignment statement of the form A(*f*)=..., where A is an array reference and *f* is an access function.
- mask(S) is true for all other statements such as conditional statements, function calls and loops and hence can be removed from the masked statement.

Hence, in the Jacobi code example, the masked version of the assignment statement S1 is [1] :

```
IF (OWNER(A(I,J)) {
  A(I,J) = (B(I,J-1) + B(I-1,J) + B(I+1,J) + B(I,J+1)) / 4
}
```

[1] This piece of code is not visible in the final transformed program in Figure 3.10 because loop bounds reduction eliminated the need for guard introduction.

A simple algorithm for generating masks for statements in a loop would be to examine each statement in turn and calculate its iteration set. If it is equivalent to the iteration set of the previous statement, both the statements may be guarded by the same mask. Otherwise, the previous mask must be terminated and a new mask created in its place. The algorithm is shown in Figure 3.9.

Algorithm for generating statement masks

for each FOR loop nest $k = l$ to m by s **do**
  Calculate iteration set of previous statement, previous_iter_set
  for each $statement_i$ in order **do**
    Calculate iteration set of $statement_i$, iter_set
    if iter_set = previous_iter_set **then**
      Insert $statement_i$ after $statement_{i-1}$
    else
      Terminate previous mask, if it exists
      Create new mask for iter_set and insert $statement_i$ inside mask
      previous_iter_set = iter_set
    endif
  endfor
endfor

Figure 3.9: Algorithm for generating statement masks

3.6.2 Message generation

The information calculated in the communication analysis phase is used to guide the message generation phase. *Non-blocking* sends and *blocking* receives are inserted for the following types of messages :

- *Loop-independent messages*

For loop-independent cross-processor dependences, our compiler inserts calls to the SEND and RECV primitives preceding the loop header. All messages are properly guarded so that owners execute SEND and recipients execute RECV. Data to be sent is calculated by building the RSD for the reference at the loop level that the message is generated. This represents the data to be sent on each iteration.

- *Loop-dependent messages*

For messages representing loop-carried dependences, the situation is more complex. To calculate the data that must be communicated, the RSD is built for each *rhs* reference at the level of the loop L carrying the dependence. If iterations of L are to be executed by all processors, the compiler inserts calls to SEND and RECV primitives inside the loop header for L, at the beginning of the loop body. If the iterations of L are to be partitioned across processors, loop-carried messages represent nothing but data synchronization. The compiler inserts calls to RECV preceding loop L, since they occur before the local iterations of L. Similarly, calls to SEND are inserted after L, since they are executed after the local iterations of L.

- *Initialization insertion*

The compiler inserts a number of initialization statements which help in initializing the MPI environment as well as gathering information at run-time about the execution environment, such as the total number of processors and the local processor number.

After the program partitioning and message generation stages, the final transformed program becomes as shown in Figure 3.10.

```
REAL A(100, 100), B(100, 100)
INTEGER I, J, K, TIME
INTEGER LB, UB
MPI_INITIALIZE ()
IF (PROC_ID() .EQ. 1)
  LB = 2
  UB = 25
ELSE IF (PROC_ID() .EQ. 2)
  LB = 26
  UB = 50
ELSE IF (PROC_ID() .EQ. 3)
  LB = 51
  UB = 75
ELSE IF (PROC_ID() .EQ. 4)
  LB = 76
  UB = 99
DO K = 1, 50
  IF (PROC_ID() .EQ. 1) SEND (B(2:99, 25), 2)
  IF (PROC_ID() .EQ. 2) RECV (B(2:99, 25), 1)
  IF (PROC_ID() .EQ. 2) SEND (B(2:99, 50), 3)
  IF (PROC_ID() .EQ. 3) RECV (B(2:99, 50), 2)
  IF (PROC_ID() .EQ. 3) SEND (B(2:99, 75), 4)
  IF (PROC_ID() .EQ. 4) RECV (B(2:99, 75), 3)
  DO J = LB, UB
    DO I = 2, 99
S1:    A(I, J) = (B(I, J-1) + B(I-1, J)
                      + B(I+1, J) + B(I, J+1)) / 4
    ENDDO
  ENDDO
  DO J = LB, UB
    DO I = 2, 99
S2:    B(I, J) = A(I, J)
    ENDDO
  ENDDO
ENDDO
MPI_FINALIZE ()
```

Figure 3.10: Final transformed program with message-passing calls

Chapter 4

Implementation

This chapter delves into the implementation of the various analysis techniques discussed in the previous chapter. The parallelizing compiler has been written in C/C++. Its target language is a C program annotated with MPI calls. The overall structure of the compiler is shown in Figure 3.2.

4.1 Front end

The SUIF compiler framework provides a compiler driver called "*scc*" for FORTRAN 77 and ANSI C programs which forms the front end of our compiler. This behaves much like the "cc" compiler in Unix. It does the lexical, syntactic and semantic analysis of the FORTRAN 77/C source program and converts it to the SUIF representation. The subsequent analysis phases work on this intermediate representation to produce the required results.

4.2 Data decomposition specifications

The data decomposition specifications (if any) are read from the user-specified decomposition file into the data structure shown in Figure 4.1.

```
struct decomposition_info {
   char name[8];               /* Array name */
   int number_dimensions;      /* Number of dimensions, max 7 */
   struct type_and_size[7];    /* Type & size for each dimension */
};
struct type_and_size {
   char distribution_type[7]; /* Distribution type, block/cyclic */
   int block_size;            /* Block size if "cyclic" specified */
};
```

Figure 4.1: Data structure for decompositions

4.3 Program Analysis

4.3.1 Dependence Analysis

Data dependence testing is a crucial step in detecting loop-level parallelism in scientific programs. The problem is equivalent to the integer linear programming problem and therefore, in general, cannot be solved efficiently. However, Dror Maydan [17] has shown that data dependence can be computed exactly and efficiently by applying a series of tests, each one exact for special case inputs. The tests are applied in the following order :

- Extended GCD Test
- Single variable per constraint test
- Acyclic test
- Simple Loop residue test
- Fourier-Motzkin test

If the input is not of the appropriate form for a particular test, the next one is applied. The advantage of using a series of tests is that it allows exact dependence analysis for a wider range of inputs. All these tests have been implemented in the SUIF compiler system and our compiler makes use of their results in the dependence analysis stage.

4.3.2 Scalar dataflow Analysis

The scalar data flow analysis is provided by a SUIF compiler pass called "*moo*". *Moo* analyzes a SUIF program for privatizable scalar variables. It essentially attaches to each FOR loop (DO construct in FORTRAN) an annotation that indicates the privatizable scalar variables. *Moo* considers a variable as privatizable if the variable is read or written in the loop and there are no upward exposed reads of the variable at the entry of the loop body, and if no definition of the variable in the loop reaches a use outside it.

4.3.3 Symbolic Analysis

Our compiler makes use of the SUIF compiler pass "*porky*" for performing the various types of analysis of the symbolic analysis phase. The command-line options to *porky* specify which transformations are to be performed. As mentioned in Section 3.5.3 of the previous chapter, the goal is to rearrange the code so as to make it easier for subsequent analysis phases to perform their analysis. For example, the forward propagation performed by *porky* tries to move as much computation as possible into the bound expressions of each loop. This may facilitate the loop bounds reduction transformation in the program partitioning phase.

4.3.4 Partitioning Analysis

The execution iteration and local index sets are represented by a class *RSD*. The RSD class definition is shown in Figure 4.2. The methods `AND` and `OR` of the RSD class provide the functionality for performing the intersection and union of two RSDs respectively. The method `is_empty()` returns true if a RSD is empty, false otherwise. Methods are also available for setting and finding the number of dimensions of a RSD. Each RSD can be thought of as being a linked list of "RSD components", one corresponding to each dimension. The data structure for a RSD component is shown in Figure 4.2.

In the partitioning analysis stage, for each array reference, the local index set for every processor is computed and is represented as an object of the RSD class. If the programmer has specified a decomposition for this array, the local index set is constructed as specified, and if no decomposition is specified, block mapping is

```
class RSD {
  private:
     int number_dimensions;      /* Number of dimensions, max 7 */
  public:
     struct RSD_component **sf;  /* RSD for each dimension      */
     RSD ();                     /* Constructor #1 */
     RSD (int num_dims);         /* Constructor #2 */
     void set_dimensions (int dims) { number_dimensions = dims; }
     int  get_dimensions () const   { return number_dimensions; }
     RSD AND (const RSD &);
     RSD OR (const RSD &);
     int is_empty () const;
     void print (FILE *fp = stdout);
};
struct RSD_component {
     int llimit;                 /* Lower limit of RSD  */
     int ulimit;                 /* Upper limit of RSD  */
     int stride;                 /* Stride of RSD       */
     struct set_format *next;    /* Pointer to next RSD */
};
```

Figure 4.2: Data structure for RSD

assumed along the first dimension and replicated mapping along other dimensions.

Next, for each array reference, the execution iteration set is computed for every processor by ANDing the RSDs represented by the corresponding local index set for that processor and the iteration set of the enclosing loops, as specified by the algorithm in Figure 3.6.

4.3.5 Communication Analysis

In this stage, we first compute the *send* and *recv* sets for each *rhs* array reference for different processor combinations using the algorithm in Figure 3.7. They are all represented as objects of the RSD class. The overall *send* and *recv* sets for the particular array assignment statement are then obtained by ORing the respective *send* and *recv* sets of each of the RSDs of the rhs array references.

4.4 Back end

The back end of the compiler has been built with the help of the SUIF Builder discussed in section 2.2 of Chapter 2.

4.4.1 Code generation

Run-time resolution is used when compile-time analysis is not applicable.

- Scalar variables are replicated on all the processors. The values of scalar variables are kept consistent across all processors by executing their defining expression on all of them. However, if the defining expression contains a reference to an array distributed across processors, only the owner processor of this reference performs the assignment. It then broadcasts this updated value to all other processors.
- Expressions in IF, DO-WHILE and WHILE loops are executed by all processors. Array assignment statements are guarded by masks to enforce owner computes rule.
- Arrays whose distributions are not specified are distributed blockwise in the first dimension.

Program partitioning and message generation

SUIF uses an abstract syntax tree (AST) representation for representing instructions and various control structures of the source program. The SUIF Builder provides explicit function calls for creating new tree nodes, tree node lists, instructions and expression trees, such as `make_tree_node()`, `make_tree_node_list()` and `make_instruction()`. In addition, new statements can be appended to tree node list elements by using calls such as `append_before()` and `append_after()`. Using these function calls, it is possible to modify existing SUIF code as well as add new SUIF code, as is required for loop bounds reduction, guard introduction and message generation.

Chapter 5

Experimental results

In this chapter, we discuss the results of the experiments conducted to measure the performance of our compiler. The compiler has been tested with a number of FORTRAN 77/C programs. Since only compile-time analysis is performed, the compiler will not parallelize programs in which arrays are created dynamically at run-time (such as by using *malloc* function call in C).

The timings are taken on Intel Pentium II workstations running RedHat Linux Version 6.0, kernel version 2.2.5-15. All the times shown are averages of atleast 15 runs. Timing is taken using the MPI_Wtime call provided by the message-passing library. MPI_Wtime is intended to be a high-resolution, wall clock that returns the time elapsed in seconds on the calling processor since the last call to this function. Hence, it is always called on processor 0, referred to as the *master processor* and is called twice, before the program's first executable statement and just before the program exits. The difference between the two gives the program computation time, assuming there are no I/O waiting periods in between.

A few of the sample programs on which the compiler was tested is given in appendix A. Jacobi.f is the Jacobi code example. Mult.c is the matrix multiplication program in C. The results of the testing are summarized in Table 5.2.

In the program `bc.f`, the second loop nest (n, i) contains an assignment to a scalar variable *neq*. This prevents loop bounds reduction and hence the improvement is not much.

It was observed that the decomposition chosen for the array references can vastly affect the computation times of programs. If the programmer wisely selects the array

Program	Lines	Loops	Decompositions	Processors	Time taken [a]	Raw time [b]
JACOBI.F	20	5	A (*, BLOCK) B (*, BLOCK)	1 2 3 4	4.4542163 2.6873997 1.5043143 1.0810553	4.048538
MULT.C	22	7	A (BLOCK, *) B (*, *) C (BLOCK, *)	1 2 3 4	10.490015 5.2711689 3.5289125 2.6510308	10.489945
BC.F	28	4	id (BLOCK, *)	1 2 3 4	0.0008724 0.0006927 0.0004762 0.0005122	0.0008469
SAMPLE.C	29	6	A (BLOCK, *) B (CYCLIC(5))	1 2 3 4	0.0024222 0.0184763 0.0418361 0.0564173	0.0021783
SAMPLE.C	29	6	A (*, BLOCK) B (BLOCK)	1 2 3 4	0.0024222 0.0013038 0.0009221 0.0007623	0.0021783
BLK4AW.F	46	7	a (BLOCK, *) ien (BLOCK, *) xl (BLOCK, *) x (BLOCK, *) rm (BLOCK, *) weis (BLOCK, *) sh (BLOCK, *) lm (BLOCK, *)	1 2 3 4	0.0586247 0.0418712 0.0357231 0.0292623	0.0561123
BLK4BW.F	59	7	b (BLOCK, *) ien (BLOCK, *) xl (BLOCK, *) x (BLOCK, *) dl (BLOCK, *) d (BLOCK, *) pp (BLOCK, *) weis (BLOCK, *) sh (BLOCK, *) lm (BLOCK, *)	1 2 3 4	0.0623542 0.0497253 0.0410012 0.0361922	0.0598123

[a] Time taken after running the programs output by our compiler (linked to MPICH library)

[b] This refers to the time taken on a uniprocessor without linking the MPICH library, measured using *gettimeofday* system call in Unix

Table 5.1: Experimental Results

decomposition by closely observing the nature of the algorithm, so that the resulting communication among the processors is minimized, the program performs better.

As an example, consider the program Sample.c in Appendix B. The second loop nest, i.e., the FOR loops of *index* and j contain two array assignment statements containing flow dependence. If the programmer specifies the data decompositions :

DECOMPOSITION A (BLOCK, *)
DECOMPOSITION B (CYCLIC(5))

the execution times on different processors are shown as in Table 5.1. It is obvious that the decomposition of A results in an attempt to parallelize along the direction of the dependence which is not possible. The processors have to execute in sequential order since each processor can go ahead with its computation only after the previous processor has finished updating its part of the array A. Hence it can be seen that with more processors, program execution time only increases and best performance is achieved on a single processor. A better decomposition set would have been :

DECOMPOSITION A (*, BLOCK)
DECOMPOSITION B (BLOCK)

in which case, the execution times would be as shown in Table 5.1. This is clearly better. BLOCK should be favoured over CYCLIC decompositions, since this leads to lesser message-passing overhead. CYCLIC should only be used when the goal is to achieve load balancing among processors.

Table 5.1 also shows that it is not possible to achieve exact linear speedup in general. In almost all the cases, an approximate n-fold speedup is seen in a n-processor system. Although only upto 4 processors have been used, the number of processors can easily be scaled up.

Chapter 6

Conclusion and future work

The results presented in the previous chapter have demonstrated that there is a significant reduction in the execution times when the same code is ported to a NOW. It can be observed that linear speedup is not obtained in general because our compiler concentrates only on exploiting loop-level parallelism. When the source program has majority of loops containing flow dependences, there is no advantage in going for NOW; the performance is worse than that obtained by sequential execution on a single processor.

6.1 Work done

The salient features of our parallelizing compiler are as follows :

- It handles block as well as cyclic decompositions specified by the programmer.
- It does the source-to-source translation, so the resulting message-passing parallel program can be directly run on a network of workstations. The program can also be run on a heterogenous NOW by supplying the appropriate command-line arguments to the MPI shell script for running MPI programs. In fact, the program can be ported to a variety of architectures, since MPI takes care of architectural differences through its portability feature.
- Dependence analysis information of the source program is made available to the user. This facilitates parallelization of loops and also provides a sort of feedback mechanism for him to know the potential array access conflicts in the program

and try and reduce these conflicts by restructuring the algorithms wherever possible. The dependency analysis output for an example program is shown at the end of Appendix B.

- The back-end of the compiler generates a MPI C program.

6.2 Future Direction

The parallelism exploited by our compiler from sequential programs is by no means the best possible. The compiler can be made even more optimal, general and robust by supplementing it with the following additional analyses :

- Performing run-time analysis for irregular computations.
- Performing program optimization
  - A program optimization phase could be incorporated between the program analysis and code generation phases of the compiler. The main goal of this phase would be to reduce communication overhead and improve program performance. Communication optimizations such as message vectorization, message pipelining, message aggregation, eliminating redundant messages and hiding communication latency by overlapping the communication and computation could be performed here.
- Improving the effeciency of the generated message-passing program.
  - Loop transformations such as loop interchange, loop tiling, loop skewing, strip mining, etc. can be implemented which will help in extracting *do-across* [1] parallelism.
- Finding the best data decomposition
  - Currently, our compiler requires the programmer's help in finding an optimal data decomposition for distributing arrays across mutiple processors. This step could be automated by the compiler, in such a way so as to maximize parallelism and minimize communication. An algorithm has been

[1] A doacross loop is one which is executed on multiple processors by assigning different iterations to different processors, and in which values assigned in one iteration are used in another iteration.

presented by Amarasinghe *et al* [1] to determine the best mapping of data and computation across multiple processors.

- Inter-procedural Analysis
  - Our compiler currently operates on each procedure individually. The algorithms can be extended to perform inter-procedural analysis.

Bibliography

[1] Amarasinghe, S., Anderson, J. M., Lam, M. S., and Lim, A. S. An Overview of a Compiler for Scalable Parallel Machines. In *Proceedings of the sixth workshop on Languages and Compilers for Parallel Computing, Available at* `http://suif.stanford.edu/papers/aall93.ps` (Portland, OR, August 1993).

[2] Amarasinghe, S. P., and Lam, M. S. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language and Design and Implementation (PLDI), Available at* `http://suif.stanford.edu/papers/amarasinghe93.ps` (Albuquerque, New Mexico, June 1993).

[3] Anderson, T. E., Culler, D. E., Patterson, D. A., and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro, Available at* `http://now.cs.berkeley.edu/Case/case.ps` *15*, 1 (February 1995), 54–64.

[4] Argonne National Laboratory. The MPICH home page. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[5] Dongarra, J. J., Hempel, R., Hey, A. J. G., and Walker, D. W. A proposal for a user-level, message passing interface in a distributed memory environment. Technical Report, TM-12231, Oak Ridge National Laboratory, Available at `http://www.epm.ornl.gov/~walker/mpi1.ps`, February 1993.

[6] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Version 1.0 Interface, May 1992.

[7] Geist, A., Begvelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. PVM : Parallel Virtual Machine, a Users' Guide and Tutorial

for Networked Parallel Computing. The MIT Press, Cambridge, Massachusetts, England, Available at `http://www.netlib.org/pvm3/book/pvm-book.html`, 1994.

[8] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H. A user's guide to PICL : a portable instrumented communication library. Technical Report, TM-11616, Oak Ridge National Laboratory, June 1991.

[9] Gelernter, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 80–112.

[10] Gropp, W. D., and Smith, B. Chameleon parallel programming tools users manual. Technical Report, ANL-93/23, Argonne National Laboratory, March 1993.

[11] Gupta, D. Compiling HPF for a Network of Workstations. *M.Tech. Thesis, Department of Computer Science and Engineering, IIT Kanpur* (April 1996).

[12] Havlak, P., and Kennedy, K. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991), 350–360.

[13] Hempel, R., Hoppe, H.-C., and Supalov, A. A Proposal for a PARMACS library interface. Technical Report, October 1992.

[14] Hiranandani, S., Kennedy, K., and Tseng, C.-W. Compiling FORTRAN D for MIMD Distributed Memory Machines. *Communications of the ACM, Available at* `http://www.cs.umd.edu/projects/cosmic/papers/cacm92.ps` *35*, 8 (August 1992), 66–80.

[15] Koelbel, C., and Mehrotra, P. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems 2*, 4 (October 1991), 440–451.

[16] Li, J., and Chen, M. Compiling communication-effecient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991), 361–376.

[17] Maydan, D. E. Accurate Analysis of array references. *Ph.D. Thesis, Department of Computer Science, Stanford University, Available at* `http://suif.stanford.edu/papers/maydan92c.ps` (September 1992).

[18] Message Passing Interface Forum. MPI: A Message Passing Interface Standard (version 1.1). Technical Report, Available at `http://www.mpi-forum.org/docs/mpi-11.ps.Z`, June 1995.

[19] nCUBE Corporation. nCUBE 2 Programmers guide, r2.0, December 1990.

[20] Parasoft Corporation. Express Version 1.0 : A Communication Environment for Parallel Computers, 1988.

[21] Pierce, P. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications* (1988), ACM Press, pp. 384–390.

[22] Rogers, A., and Pingali, K. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 conference on Programming Language Design and Implementation* (Portland, OR, June 1989), pp. 69–80.

[23] Skjellum, A., Smith, S., Leung, A., and Morari, M. The Zipcode message passing system. Technical Report, Lawrence Livermore National Laboratory, September 1992.

[24] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI - The Complete Reference, Volume 1.* The MIT Press, 1998.

[25] Stanford Compiler Group. The SUIF Library. `http://suif.stanford.edu/suif/suif1`, 1994.

[26] Tseng, C.-W. An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines. *Ph.D. Thesis, Department of Computer Science, Rice University, Published as Rice COMP TR93-199, Available at* `http://www.cs.umd.edu/projects/cosmic/papers/thesis.ps` (January 1993).

[27] Wolfe, M. J. *High-performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[28] Zima, H., Bast, H. J., and Gerndt, M. SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing 6* (1988), 1–18.

[29] Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley Publishing Company, 1990.

Appendix A

User Manual

A.1 The parallelizing compiler

The generation of the parallel MPI program from the given source program has been done with the help of a shell script, which is available as a command name *parallelize*. This is basically done so that the user is insulated from the details of interfacing the source program with SUIF, such as passing the source program through the SUIF front end and performing the various compiler passes on it.

A.2 Usage

The parallelizing compiler is invoked by the command :

```
$ parallelize [options] SUIF-file
   (where SUIF-file is obtained from the source program after running
    the SUIF compiler driver scc)
Options available are :
-n proc         number of processors
-f filename     filename containing user-specified decompositions
-o filename.c   Output message-passing program in C
-h              help
```

A.2.1 Description of options

- `-n proc`
  - The number of processors/machines constituting the NOW. *parallelize* takes the number of processors either from this command-line option or from the environment variable NUM_PROC. If none of these are specified, it assumes that the default number of processors is 4.
- `-f filename`
  - A file, input to the parallelizing compiler, which contains the array decomposition specifications. It should consist of a sequence of DISTRIBUTE statements, one per line, corresponding to the desired decomposition pattern of an array reference, as discussed in Section 3.3 of Chapter 3.
- `-o filename.c`
  - The parallel C program, containing MPI calls, output by the parallelizing compiler. If `-o` option is not specified, output is written to "parcode.c" by default.

A.3 Hardware and Software Requirements

- **Platform** : no specific requirement; the software was developed on a Intel Pentium II machine under RedHat Linux 6.0 (Kernel Ver. 2.2.5-15).
- **Software needed** :
  1. MPICH library, version 1.1.2 or later, should be installed on every machine of the NOW (Available at `http://www-unix.mcs.anl.gov/mpi/`).
  2. SUIF compiler system, version 1.3.0.1 or later but below 2.0 (Available at `http://suif.stanford.edu/suif/suif1/`).
  3. F77 and I77 libraries for FORTRAN.

A.4 Steps in compiling to the NOW

1. Set the environment variable NUM_PROC, if desired. For example, to set it to 3, do the following :

   ```
   for ksh
   $ export NUM_PROC=3
   for csh
   $ setenv NUM_PROC 3
   ```

2. Run the SUIF compiler driver *scc* to convert the source program to a SUIF file (denoted by ".spd"). For example, after running scc on "input.c", the output is "input.spd" :

   ```
   $ scc -V -.spd input.c
   ```

   If you need to parallelize a collection of subroutines, say 1.f, 2.f and 3.f, first run *scc* as described above on each of them to get the SUIF files, say 1.spd, 2.spd and 3.spd. Then, combine these SUIF files into a file set using *linksuif* and them merge them into a single SUIF file using *mergesuif*.

   ```
   $ linksuif 1.spd 1.out 2.spd 2.out 3.spd 3.out
   $ mergesuif 1.out 2.out 3.out input.spd
   ```

3. Assuming that after step 1, the SUIF file equivalent to the source program is "input.spd", invoke the command "parallelize input.spd", with any of the options described above, to get the MPI program in C. Let us call it "parcode.c".

4. Compile and link "parcode.c" with the MPICH library on each workstation. This can be done using the mpicc compiler. Ensure that the path of the resulting executable program is the same on every machine.

   ```
   For C programs :
   $ mpicc -o parcode parcode.c
   For FORTRAN 77 programs :
   $ mpicc -lF77 -lI77 -o parcode parcode.c
   ```

5. Run the MPI program on the NOW.

```
$ mpirun parcode
```

A.5 Additional notes

- MPICH requires a process startup mechanism for starting up processes on a network of workstations. Normally *rsh* is used (under Unix), but certain permissions have to be set up on the participating machines, such as creation of *.rhosts* file. If this is undesirable on your network, the simplest alternative is to use *ssh*, a secure shell. For more details on configuring *ssh* to work properly with MPICH, refer to the Installation Guide of MPICH (available with the MPICH distribution).

- The configuration of the virtual machine is defined in the file `mpich/util/machines/machines.XXX` where XXX represents the architecture of the underlying machines. Hence, the user may specify the host names of the machines constituting the NOW in this file.

- For instructions on how to run MPI programs on a heterogenous NOW, refer to the User's Guide of MPICH.

Appendix B

Sample Programs

JACOBI.F

```
   PROGRAM JACOBI
   REAL A(500, 500), B(500, 500)
   INTEGER I, J, K, TIME
   TIME = 100

   DO 50 K = 1, TIME
     DO 20 J = 2, 499
       DO 10 I = 2, 499
         A(I, J) = (B(I,J-1) + B(I-1,J) + B(I+1,J) + B(I,J+1)) / 4
10     CONTINUE
20   CONTINUE

     DO 40 J = 2, 499
       DO 30 I = 2, 499
         B(I, J) = A(I, J)
30     CONTINUE
40   CONTINUE
50 CONTINUE

   END
```

MULT.C

```
#include <stdio.h>
void main (int argc, char *argv[])
{
  int A[300][300], B[300][300], C[300][300];
  int i, j, k;
  for (i=0;i<300;i++)
    for (j=0;j<300;j++)
    {
      A[i][j] = 1;
      B[i][j] = 2;
      C[i][j] = 0;
    }

  for (i=0;i<300;i++)
    for (j=0;j<300;j++)
      for (k=0;k<300;k++)
        C[i][j] = C[i][j] + A[i][k]*B[k][j];

  for (i=0;i<300;i++)
    for (j=0;j<300;j++)
      printf ("C[%d][%d]=%d\n", i, j, C[i][j]);
}
```

BC.F

```
      subroutine bc (id, d, neq)
      parameter (mrng=21)
      integer rng(mrng)
      real rs(mrng)
      common /crng/ nrng, rng, rs
      common /cinfo/ ndof, nsd, nn, ne
      integer id(ndof,1)
      real d(ndof,1)

      do 10 idof=1, ndof
        do 10 inode=1,nn
        id (idof, inode) = 0
 10   continue

c.....establish equation numbers
      neq = 0
      do 20 n=1,nn
        do 20 i=1,ndof
          if (id(i,n).ne.0) then
            id(i,n) = 0
          else
            neq = neq + 1
            id(i,n) = neq
          end if
 100  continue

      return
      end
```

SAMPLE.C

```
#include <stdio.h>
int A[30][30];
int B[30];
void main(int argc, char *argv[])
{
    int i,j,index;
    for (i=0;i<10; i++)
    {
      for (j=0;j<30;j++)
       {
         A[i][j]=0;
         B[i]=0;
       }
    }
    for (index=3;index<30;index++)
      for (j=27;j>0;j--)
       {
         A[index][j] = A[index-1][j] + A[index-2][j] + A[index-3][j]+1;
         B[index] = B[index-1] + 1;
       }

    for (i=0;i<30;i++)
      for (j=0;j<30;j++)
       {
         if (j==0) printf ("%d::%d::%d,", i, B[i],A[i][j]);
         else if (j<29) printf("%d,", A[i][j]);
         else printf("%d\n", A[i][j]);
       }
}
```

```
      subroutine blk4aw (a, ien, x, lm)
      parameter (nen=4)
      parameter (nee=4)
      common /cblk/ neq, length
      common /cinfo/ ndof, nsd, numnp, numel
      integer ien(nen,1), lm(nen,1)
      real a(1), x(nsd,1)
      real gaus(2,4), weis(4)
      real rm(nen), xl(2,nen)
      real sh(0:2,nen)
      real det, eff

      isymm = 0
      do 10 i=1, neq
        a(i)=0.0
 10   continue

      do 200 n= 1, numel

        do 20 inen = 1,nen
          inode = ien(inen,n)
c.......... localize co-ordinates
           do 30 isd  =1,nsd
            xl(isd,inen) = x(isd,inode)
 30       continue
 20     continue

        do 40 iee = 1, nee
          rm(iee) = 0.0e0
 40     continue
```

```
      do 120 k=1,ngaus
        eff = det * weis(k)
        do 110 nb=1,nen
           rm(nb) = rm(nb) + sh(0,nb) * eff
110     continue
120   continue

      a(lm(1,n)) = a(lm(1,n)) + rm(1)
      a(lm(2,n)) = a(lm(2,n)) + rm(2)
      a(lm(3,n)) = a(lm(3,n)) + rm(3)
      a(lm(4,n)) = a(lm(4,n)) + rm(4)

200  continue
     return
     end
```

```
      subroutine blk4bw (b, ien, x, d, pa ,lm)
      parameter (nen=4)
      parameter (nee=4)
      common /cblk/ neq, length
      common /cinfo/ ndof, nsd, numnp, numel
      real b(1), x(nsd,1), d(nsd,1), pa(1)
      integer ien(nen,1), lm(nen,1)
      real gaus(2,4), weis(4)
      real pp(nen), xl(2,nen), dl(2,nen)
      real sh(0:2,nen)
      real xr, det, eff, w

      do 10 i=1,neq
        b(i) = 0.0
 10   continue

      do 200 n=1, numel

        do 20 inen =1,nen
          inode = ien(inen,n)
c.......... localize co-ordinates
          do 30 isd  =1,nsd
            xl(isd,inen) = x(isd,inode)
 30       continue
c.......... localize dof's
          do 40 idof =1,nsd
           dl(idof,inen) = d(idof,inode)
 40       continue
 20     continue

        do 50 iee =1, nee
```

```
          pp(iee) = 0.0e0
50      continue

        do 60 k=1,ngaus
          eff = det * weis(k)
            w    = sh(1,1) * dl(2,1)
     &           + sh(1,2) * dl(2,2)
     &           + sh(1,3) * dl(2,3)
     &           + sh(1,4) * dl(2,4)
     &           - sh(2,1) * dl(1,1)
     &           - sh(2,2) * dl(1,2)
     &           - sh(2,3) * dl(1,3)
     &           - sh(2,4) * dl(1,4)

            pp(1) = pp(1) + sh(0,1) * w * eff
            pp(2) = pp(2) + sh(0,2) * w * eff
            pp(3) = pp(3) + sh(0,3) * w * eff
            pp(4) = pp(4) + sh(0,4) * w * eff
60      continue

        b(lm(1,n)) = b(lm(1,n)) + pp(1)
        b(lm(2,n)) = b(lm(2,n)) + pp(2)
        b(lm(3,n)) = b(lm(3,n)) + pp(3)
        b(lm(4,n)) = b(lm(4,n)) + pp(4)

200   continue
      return
      end
```

```
Dependency information generated by compiler for SAMPLE.C

=======main=======
----------------------------------------
For loop i (Outermost)
For loop j
*******************************
A[i,j] on line 11
*******************************
B[i] on line 12

   Output dependence:
   ( 0 * )
----------------------------------------
For loop index (Outermost)
For loop j
*******************************
A[index,j] on line 18

   Anti/Flow dependence:
   [1] A[index-1,j] on line 18
       ( -1 0 )
   [2] A[index-2,j] on line 18
       ( -2 0 )
   [3] A[index-3,j] on line 18
       ( -3 0 )
*******************************
B[index] on line 19

   Anti/Flow dependence:
   [1] B[index-1] on line 19
       ( -1 )
```