SIMULTANEOUS EXPLOITATION OF TASK AND DATA PARALLELISM
IN REGULAR SCIENTIFIC APPLICATIONS

BY

SHANKAR RAMASWAMY

B.S., University of Delhi, 1988
M.E., Indian Institute of Science, 1991

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

SIMULTANEOUS EXPLOITATION OF TASK AND DATA PARALLELISM
IN REGULAR SCIENTIFIC APPLICATIONS

Shankar Ramaswamy, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1996
Prithviraj Banerjee, Advisor

Distributed Memory Multicomputers (DMMs) such as the IBM SP-2, the Intel Paragon and the Thinking Machines CM-5 offer significant advantages over shared memory multiprocessors in terms of cost and scalability. Unfortunately, the utilization of all the available computational power in these machines involves a tremendous programming effort on the part of users, which creates a need for sophisticated compiler and run-time support for distributed memory machines. In this thesis we explore a new compiler optimization for regular scientific applications - the simultaneous exploitation of task and data parallelism. Our optimization is implemented as part of the PARADIGM HPF compiler framework and as part of a MATLAB compiler framework we have developed. The intuitive idea behind the optimization is the use of task parallelism to control the degree of data parallelism of individual tasks. The reason this provides increased performance is that data parallelism provides diminishing returns as the number of processors used is increased. By controlling the number of processors used for each data parallel task in an application and by concurrently executing these tasks, we make program execution more efficient and therefore faster. A practical implementation of a task and data parallel scheme of execution for an application on a distributed memory multicomputer also involves data redistribution. This data redistribution causes an overhead. However, as our experimental results show, this overhead is not a problem; execution of a program using task and data parallelism together can be significantly faster than its execution using data parallelism alone. This makes our proposed optimization practical and extremely useful.

To my wife, Meena and my parents, Nirmala and Ramaswamy

ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor Professor Prith Banerjee for all the encouragement and support he has provided over the years I have spent at Illinois. He has been a great friend and mentor.

I would also like to thank the members of my committee – Professors Wen-mei Hwu, Constantine Polychronopoulos, and David Padua for their invaluable comments and suggestions.

Sachin Sapatnekar, Eugene Hodges and John Chandy have all helped me at various points in my research; I am thankful to them for their help. Dan Palermo has been my personal LaTeX and Unix help desk over the years – I am very grateful for his help. Ernesto Su and Amber Roy-Chowdhury have been great office mates. I will always remember the thoroughly enjoyable discussions – technical and otherwise – with Antonio Lain during our coffee breaks. Carolin Tschopp has been most helpful with all administrative matters.

My parents Nirmala and Ramaswamy have been a constant source of encouragement and love without which I would not have embarked on this career path. Finally, I would like to thank my wife Meena for all her love and support. In spite of her own busy schedule as a graduate student, she has always been there for me when I have needed her most.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Description

Distributed Memory Multicomputers (DMMs) such as the IBM SP-2, the Intel Paragon and the Thinking Machines CM-5 offer significant advantages over shared memory multiprocessors in terms of cost and scalability. Unfortunately, the utilization of all the available computational power in these machines involves a tremendous programming effort on the part of users. This creates a need for sophisticated compiler and run-time support for distributed memory machines. In this thesis we explore a new compiler optimization for regular scientific applications - the simultaneous exploitation of task and data parallelism.

Scientific applications are typically subdivided into two major classes – *Regular* and *Irregular* applications. In regular applications, the data structures used are dense arrays and the accesses to these data structures can be characterized well at compile time. In irregular applications, some of the data structures used may be sparse arrays whose structure can only be determined at the time of program execution. Our work focuses on the domain of regular applications.

With respect to parallelism, applications can have either *Data* or *Task* parallelism or a mix of both types. For the purpose of our work, data parallelism is defined to be the parallelism

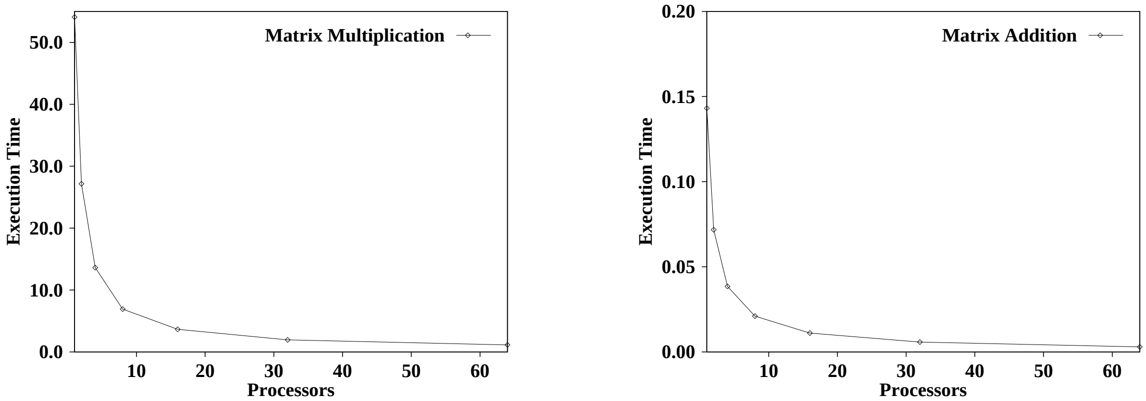
```
PROGRAM COMPLEX_MATRIX_MULTIPLY

CALL MATRIXMULTIPLY(AReal,BReal,Temp1)
CALL MATRIXMULTIPLY(AImag,BImag,Temp2)
CALL MATRIXMULTIPLY(AReal,BImag,Temp3)
CALL MATRIXMULTIPLY(AImag,BReal,Temp4)

CALL MATRIXSUBTRACT(Temp1,Temp2,CReal)
CALL MATRIXADD(Temp3,Temp4,CImag)

END
```

Figure 1.1 Multiplication of Complex Matrices



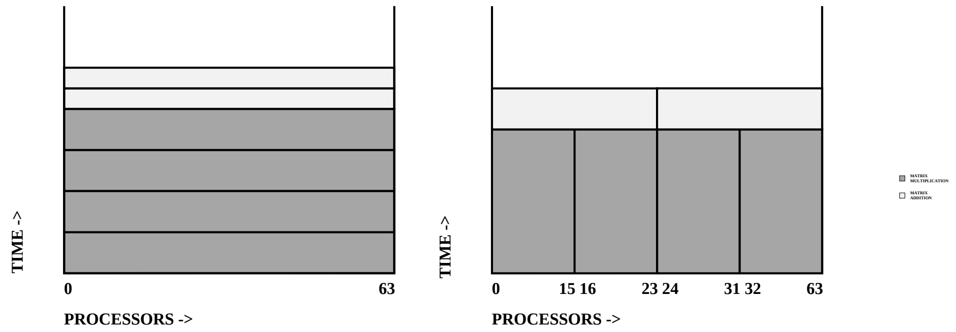
(a) Matrix Multiplication (b) Matrix Addition

Figure 1.2 Execution Time for Matrix Multiplication and Addition on the CM-5

obtained by concurrent computation using different portions of a set of data structures. On the other hand, task parallelism is defined to be the parallelism obtained by concurrent computation using different sets of data structures. The computation on each set of data structures is called a task. Scientific applications often have a mix of both types of parallelism, i.e., there are tasks that can be executed concurrently as well as each task can be executed in a data parallel fashion. However, the degree of task parallelism in scientific applications is fairly small while data parallelism is fairly abundant.

An example of a scientific program is shown in Figure 1.1. The program corresponds to the multiplication of two complex matrices (`(AReal,AImag)` and `(BReal,BImag)`) to produce an output complex matrix `(CReal,CImag)`. The matrix `CReal` is produced by multiplying `AReal` with `BReal` and `AImag` with `BImag` and taking their difference. The matrix `CImag` is produced by multiplying `AReal` with `BImag` and `AImag` with `BReal` and taking their sum. There are six tasks in the program - four matrix multiplications, one matrix addition and one matrix subtraction. Figure 1.2 plots the data parallel execution times for the two basic tasks in the program - matrix multiplication and matrix addition/subtraction. The times were measured for 256×256 element input matrices using a Thinking Machines CM-5.

Now, given 64 processors, a pure data parallel scheme of execution for our example program would involve executing all the tasks one after the other using all 64 processors. Figure 1.3(a) illustrates this scheme of execution. The pure data parallel execution scheme finishes executing the program in 8.5 seconds.



(a) Pure Data Parallel (b) Task and Data Parallel

Figure 1.3 Execution Schemes for Complex Matrix Multiplication Program

On the other hand, if we use a pure task parallel scheme of execution for the program, we would first execute the four matrix multiplications concurrently using one processor each and then execute the matrix addition and matrix subtraction concurrently using one processor each. The pure task parallel execution scheme finishes in 54.2 seconds. As mentioned before, the degree of task parallelism in scientific applications is small, which typically makes pure data parallel execution much better as compared to a pure task parallel execution. Therefore, pure task parallel execution schemes are not important and shall be ignored in the rest of this thesis.

A third possible scheme of execution for the program is one that uses task and data parallelism together. Figure 1.3(b) illustrates such a scheme. In this scheme the four matrix multiplications execute concurrently using 16 processors each, then the matrix addition and subtraction execute concurrently using 32 processors each. Using this mixed parallelism scheme, the program executes in 7.5 seconds. Clearly, this scheme of execution is the fastest for the given example program.

Our example illustrates the benefits to be obtained by using the small degree of task parallelism available in scientific applications together with the available data parallelism. Task parallelism can be used to control the degree of data parallelism of individual tasks. The reason for an improvement in the execution time as compared to the use of pure data parallelism is the diminishing returns from data parallelism as the number of processors used for a task is increased. The execution time plot for the matrix multiplication in Figure 1.2 clearly shows this. When we increase the number of processors used for a multiply from 16 to 64 (a factor of 4), the execution time for the matrix multiplication does not drop proportionately, it reduces from 3.65 seconds to 1.14 seconds which is a factor of just 3.2.

1.2 Thesis Contributions

In considering the problem of simultaneous exploitation of task and data parallelism on distributed memory multicomputers, our thesis makes the following contributions:

- Our research on simultaneous exploitation of task and data parallelism has been carried out in the context of parallel languages by extending the High Performance Fortran (HPF) [1],[2] standard and extending the PARADIGM compiler [3] to automatically exploit task and data parallelism in extended HPF programs. Section 1.3 contains a brief discussion of HPF and the reason for adding extensions to HPF.

- Our research has also considered the problem of exploiting task and data parallelism in the context of parallel libraries. We have implemented a compiler that translates MATLAB [4] programs to parallel programs that are based on the ScaLAPACK [5],[6] parallel library. During translation, the compiler exploits available task and data parallelism. A brief discussion of MATLAB and ScaLAPACK is included in Section 1.4.

- The issues involved in automatically exploiting task and data parallelism in extended HPF or MATLAB applications are the same and can be subdivided into the following three problems:

  (1) Detection and representation of task and data parallelism available in the application. The representation we use is the *Macro Dataflow Graph (MDG)*. We have developed algorithms for the automatic extraction of an MDG from an extended HPF or MATLAB program.

  (2) Using this information to decide on a scheme of execution for the application. For this, we use an *Allocation and Scheduling* approach. Allocation decides on the number of processors to be used for each task in the program and scheduling decides on the order of execution for tasks. The goal of the allocation and scheduling algorithm is to minimize the execution time for the given program.

  (3) Implementing the execution scheme with necessary run-time library support. The most important type of run-time support required is *Data Redistribution*. The exploitation of task and data parallelism could necessitate having tasks execute on

subsets of processors. Arrays that are written to by a task executing on one subset and read by a task executing on another subset have to be redistributed.

- We have evaluated the benefits of our optimization of using task and data parallelism together for a set of benchmark programs.

1.3 High Performance Fortran (HPF)

Numerous research efforts have proposed language extensions to Fortran in order to ease programming multicomputers; the most prominent one has been the High Performance Fortran (HPF) language standardization [1],[2]. The most significant feature of HPF is the data distribution directives it provides. These directives enable the programmer to specify a layout scheme for all the arrays in a program. Given such a data layout scheme, an HPF compiler generates a data parallel program for any given target architecture. In its current form HPF does not provide for specification of task parallelism. Researchers have proposed extensions to HPF for the specification of task parallelism [7],[8]; we have adopted some of these extensions for our HPF compiler. Chapter 3 discusses these extensions in more detail.

A number of compilers for HPF have been proposed; these include the PARADIGM compiler from the University of Illinois [3], FORTRAN-D compiler from Rice University [9], the SUIF compiler from Stanford [10], the pHPF compiler from International Business Machines [11], the High Performance Fortran 90 compiler from Digital Equipment Corporation [12], the pgHPF compiler from the Portland Group Inc. [13], the xHPF compiler from Applied Parallel Research [14], the SUPERB compiler from the University of Vienna [15], and the FORTRAN-90D/HPF compiler from Syracuse University [16].

The PARADIGM compiler project at the University of Illinois [3],[17] aims at combining all of the following aspects of compiling for distributed memory machines into a single framework:

- Automatic data distribution for regular scientific applications [18]-[21].
- Computation partitioning, communication generation and communication optimization for regular scientific applications [22]- [27].
- Support for irregular scientific applications using a combination of compile-time analysis and a run-time library [28]-[30].

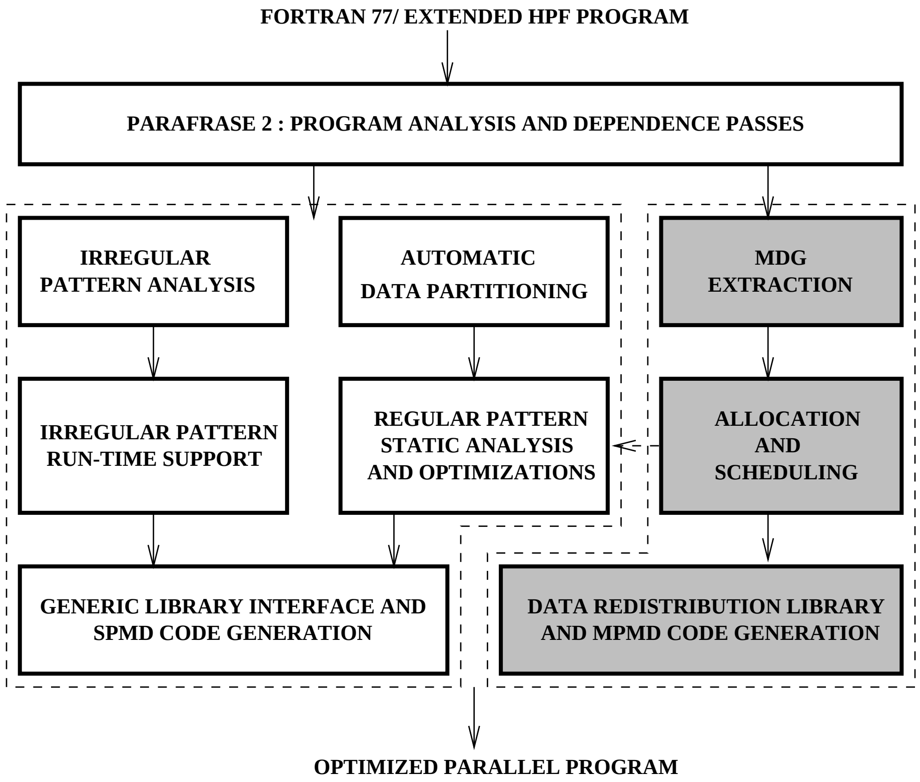


Figure 1.4 Organization of the PARADIGM Compiler

- Simultaneous exploitation of task and data parallelism in regular scientific applications [31]- [36].

Figure 1.4 illustrates the organization of the PARADIGM compiler. The compiler currently accepts as input an extended HPF or Fortran 77 application. The research detailed in this thesis is implemented in the shaded blocks and adds the capability of simultaneously exploiting task and data parallelism in extended HPF applications. We have added extensions to HPF for expressing task parallelism; these extensions are discussed in Chapter 3. The modules we have added to PARADIGM extract the MDG structure from an extended HPF program and use it to exploit task and data parallelism together. Our code generation module generates Multiple Program Multiple Data (MPMD) code that contains calls to data parallel codes for individual tasks and the data redistribution library. Data parallel code for individual tasks is generated using the regular pattern analysis and optimizations module and the Single Program Multiple Data (SPMD) code generation module in the PARADIGM framework. We have augmented the PARADIGM run-time system to include data redistribution routines.

1.4 MATLAB and ScaLAPACK

MATLAB [4] is a commercially available software package from MathWorks Inc. that is tailored for rapid prototyping of applications by scientists and engineers. MATLAB is used in research and industry for both general purpose as well as special purpose numerical computations. The package consists of an interpreter and a large library of intrinsic routines and operators. Matrices are basic data types in the environment and can be used directly as operands or function call parameters. Using the intrinsics, many toolboxes have been developed for specific application domains by MathWorks, third-party vendors and university researchers. All of these have resulted in the MATLAB package being widely used.

The ScaLAPACK [5],[6] parallel library of linear algebra routines for distributed memory multicomputers has evolved from the LAPACK [37], EISPACK [38] and LINPACK [39] libraries. It is aimed at providing scalable performance on distributed memory machines. The routines use block cyclic data distributions for arrays to obtain good performance and minimize data movement between various levels in the memory hierarchy. The ScaLAPACK library uses the Basic Linear Algebra Communication Subroutines (BLACS) [40] for all interprocess communication. The latest version of BLACS supports concurrent execution of ScaLAPACK routines on different subpartitions of a machine. This makes for easy generation of task and data parallel codes using ScaLAPACK.

The core computational intrinsics in MATLAB started out being based on the LINPACK and EISPACK projects. Since ScaLAPACK is a parallel version of these libraries aimed at distributed memory machines, we thought that translating MATLAB programs to parallel programs based on ScaLAPACK would be a natural way of parallelizing MATLAB programs. In addition, in performing the translation automatically, we are also able to apply optimizations such as the exploitation of task and data parallelism. The organization of our MATLAB compiler is shown in Figure 1.5. The compiler parses in an extended MATLAB program (the extensions are discussed in Chapter 3) and extracts the MDG structure from it to exploit task and data parallelism together. The code generation module generates MPMD code that contains calls to ScaLAPACK/BLACS library routines and the data redistribution library routines.

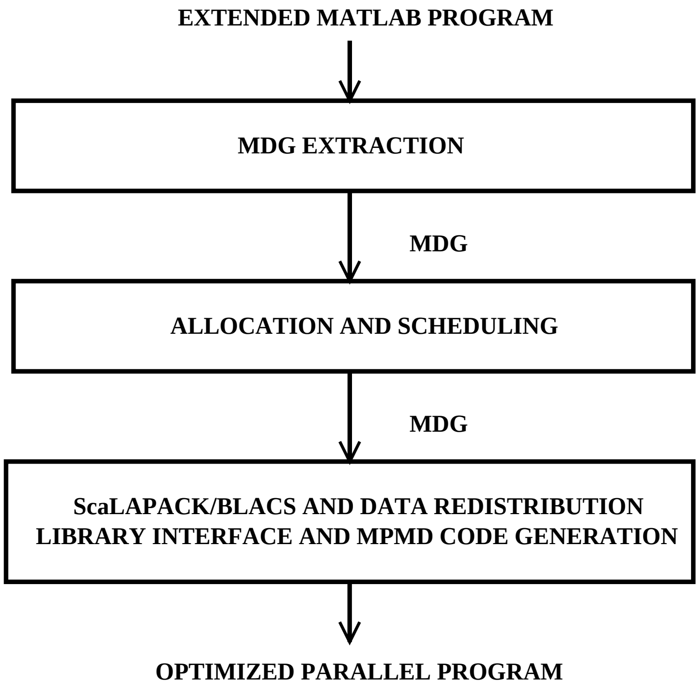


Figure 1.5 Organization of the MATLAB Compiler

1.5 Thesis Outline

Chapter 2 provides an overview of important related work. Chapter 3 describes the Macro Dataflow Graph (MDG) representation in detail and discusses its automatic extraction from extended HPF and MATLAB programs. In Chapter 4 we discuss our allocation and scheduling algorithms. Chapter 5 presents the details of our run-time support for data redistribution. Results demonstrating the usefulness of our optimization are discussed in Chapter 6. Finally, Chapter 7 summarizes the implications of our work and outlines possible future work.

CHAPTER 2

RELATED WORK

2.1 Task and Data Parallelism

Recently, there has been growing interest in simultaneous exploitation of task and data parallelism in Fortran applications. Some of the significant research efforts in the area include the Fx compiler project at Carnegie Mellon University, the Fortran-M/HPF project at Argonne National Laboratory and the Parafrase project at the University of Illinois.

The Fx compiler project [8],[41] targets a class of applications [42] that process continuous streams of data sets such as images from a video camera. The processing for each data set is essentially a data parallel computation. Task parallelism is derived by operating concurrently on different data sets. Input to the Fx compiler is an extended form of HPF. The compiler implements allocation and scheduling algorithms that are aimed at the class of applications it targets. As we will see later, our compiler targets a more general class of applications than the Fx compiler and uses more sophisticated allocation and scheduling algorithms.

Fortran-M [43] was initially conceived as an extension to Fortran for expressing task parallelism. Recently, it has been integrated with HPF [7] in order to enable the simultaneous exploitation of task and data parallelism. The Fortran-M/HPF project targets a class of applications that have a few extremely heavyweight data parallel tasks continuously interacting with each other. The Fortran-M/HPF compiler does not implement any algorithms for automatic allocation and scheduling, the user has to provide this information.

The Parafrase compiler project [44] parallelizes Fortran applications for shared memory machines. It analyzes an input Fortran program and constructs a Hierarchical Task Graph (HTG) representation [45],[46] for the program. The HTG representation captures parallelism information at all levels of granularity. Based on the HTG representation, an *Autoscheduling* [47]-[49] technique has been proposed for exploiting task and data parallelism dynamically during program execution. This technique involves the selection of an appropriate granularity for executing the various tasks in a program during execution, thereby adapting to the load on the underlying machine. The autoscheduling technique is well-suited for shared memory machines. However, on distributed memory machines, we feel that the granularity of tasks will have to be large for a run-time technique such as autoscheduling to work well. The reason for this is the high cost of communication and synchronization on distributed memory machines.

2.2 Extensions to HPF

As mentioned before, the HPF standard does not provide directives for expressing task parallelism at this time. The HPF forum is currently considering including a set of directives for this purpose. Meanwhile, researchers have independently proposed directives for expressing task parallelism in HPF programs:

- Foster et al. [7] have proposed the `PROCESSES` statement as an extension to HPF. The body of the `PROCESSES` statement contains independent calls to a set of data parallel HPF subroutines. The call to each HPF subroutine must be annotated by the number of processors to be used for the routine (allocation). Scheduling of routines must also be done explicitly using the `CHANNEL` statement. This statement establishes a virtual communication interface between a pair of routines resulting in a precedence relationship between the routines.

- Gross, O'Halloran and Subhlok [8] have proposed the notion of *Parallel Sections* in a given HPF application. These sections are demarcated using the `PARALLEL` directive. A parallel section can contain HPF data parallel subroutine calls and simple `DO`loops. Each HPF data parallel subroutine call is annotated using the `IN` and `OUT` directives. These directives provide the compiler with information about the side effects of the subroutines called and are helpful is deciding on precedence constraints between subroutine calls. Allocation of processors to HPF data parallel routines can be explicitly managed using the `PROCESSOR` and `ORIGIN` directives. Scheduling is based on precedence constraints between subroutine calls and is done implicitly by the compiler.

For our research, we have essentially adopted a modified version of the directives proposed by Gross, O'Halloran and Subhlok. The major difference is that we allow for `IF` and `DO WHILE` statements inside a parallel section. Further, our compiler performs allocation and scheduling automatically without any user intervention.

2.3 MATLAB Compilers

De Rose et al. [50],[51] are working on a MATLAB compiler to facilitate rapid prototyping of numerical libraries and programs on shared memory machines. Their compiler performs detailed analysis of a given MATLAB program and translates it into a Fortran 90 program augmented with directives for parallelization and calls to any available parallel library routines. At this point the compiler from De Rose et al. exploits only data parallelism. Our MATLAB compiler targets distributed memory machines and exploits both task and data parallelism.

2.4 Macro Dataflow Graph

The hierarchical Macro Dataflow Graph (MDG) representation we use for programs is very similar to the Hierarchical Task Graph (HTG) representation of Girkar and Polychronopoulos [45],[46]. The HTG representation is very powerful and captures parallelism information at all levels of granularity for Fortran programs. The MDG abstracts much of this information and is well-suited for our allocation and scheduling algorithms.

2.5 Allocation and Scheduling

The basic problem of optimally scheduling a set of nodes with precedence constraints on a p processor system when each node uses just one processor has been shown to be NP-complete by Lenstra and Kan [52]. Further treatment on this topic can also be found in the book by Garey and Johnson [53]. The problem of allocation and scheduling we are considering in this thesis is considerably harder than the simple scheduling problem. There have been two major approaches to the approximate solution of the allocation and scheduling problem. The first has been a *Bottom-up* approach like those used by Sarkar [54] and by Yang and Gerasoulis [55],[56]. A bottom-up approach considers nodes in the MDG to be lightweight (in terms of computation requirements) with each node using only one processor (an explicit allocation is not done). The algorithms of [54]-[56] use clustering on the nodes of the MDG to form larger nodes during the construction of a schedule. The second approach to allocation and scheduling is a *Top-down* approach like ours. Other researchers who have used top-down allocation and scheduling include Prasanna and Agarwal [57], Prasanna et al. [58], Belkhale and Banerjee [31],[32], and Subhlok et al. [59],[60], Top-down approaches start with the assumption of heavyweight nodes (again, in terms of computation requirements) in the MDG and explicitly allocate processors to nodes during the process of constructing an optimal schedule. Top-down approaches take a more global view of the problem than bottom-up approaches. Therefore, they may be able to perform better optimizations.

The differences between the top-down approaches of other researchers and our approach are significant. A primary difference between our approach and all other approaches stems from our ability to handle program control flow constructs in the MDG. There are also specific differences with respect to each of the related approaches. The methods presented in [57],[58] do not consider data transfer costs between nodes of the MDG. They also make simplifying assumptions about the type of MDGs handled and the processing cost model used. We do not make any assumptions for our MDGs and use very realistic cost models. The work in [31],[32] also does not consider the effects of nonzero data transfer costs. Their allocation and scheduling algorithms are similar to the ones we use. The research discussed in [59],[60] considers allocation and scheduling for a specific class of problems that process continuous streams of data sets. The computation for each data set has a simple tree-structured MDG for all their benchmark programs [42]. It is not clear how the proposed allocation and scheduling heuristics would work for more general, non-tree MDGs. Our methods are able to handle all MDGs in a uniform manner.

2.6 Data Redistribution

It is possible for one to argue that data redistribution can be performed using multicomputer compiler techniques such as those outlined in [9],[24],[26],[61]. These techniques generate the communication required for any program statement to execute correctly given the distributions of the arrays involved in the statement. One could now use a statement of the form `B=A` where `B` is distributed according to the target distribution and `A` is distributed according to the source distribution. However, it is not clear that any of the currently implemented compilers have efficient techniques to handle all the regular distributions possible for `A` and `B`. Considering only the problem of data redistribution allows us to use many optimizations that may not be applicable in a general compiler framework. Another major obstacle in trying to use current compiler techniques is that they cannot handle the given statement if `A` and `B` are distributed on subsets (possibly overlapping) of processors. This is essential for our work.

The work by Agarwal, Sussman and Saltz [62] provides run-time support for redistributions. They construct schedules for redistribution during program execution and reuse schedules if a particular redistribution pattern occurs more than once. However, this work handles neither the `CYCLIC` or `BLOCKCYCLIC` type of distributions nor does it consider arbitrary source and target processor sets.

The work by Thakur, Choudhary and Fox [63] considers a library for changing the distribution of arrays on a given set of processors. The methods proposed treat possible source-target distributions in a pairwise manner. This prevents them from handling very general source-target distributions in an efficient manner. Further, [63] propose an expensive solution for multidimensional redistributions. They consider such redistributions to be composed of a series of one-dimensional redistributions, which can lead to a considerable amount of unnecessary communication.

Coelho and Ancourt [64] have proposed a general theoretical framework for data redistribution and are currently developing an implementation of this framework. Our algorithms are more practical and target frequently occurring cases of data redistribution in a highly efficient manner.

CHAPTER 3

THE MACRO DATAFLOW GRAPH

3.1 Introduction

To exploit available task and data parallelism for an application, we require a program representation that captures information about both types of parallelism. The program representation we use is called the Macro Dataflow Graph (MDG) and is very similar to the Hierarchical Task Graph representation proposed by Girkar and Polychronopoulos [45],[46]. Our allocation and scheduling algorithms discussed in Chapter 4 use information in the MDG to decide on an execution scheme for a given application.

Some of the important properties of the MDG are

- It is a weighted directed acyclic graph.
- Nodes represent computation and edges represent precedence constraints.
- There are two distinguished nodes called `START` and `STOP` in the MDG. `START` precedes all other nodes and `STOP` succeeds all other nodes.
- The MDG is hierarchical in nature, i.e., nodes in an MDG may actually contain MDGs themselves.

The basic types of nodes in an MDG are

- Simple (`S`) nodes that correspond to a basic data parallel task in the given application. At the lowest level of the hierarchy in an MDG, all nodes are `S` nodes.
- Loop (`L`) nodes that correspond to loop constructs in the given application. Loops can be `FOR` loops or `WHILE` loops. An `L` node contains an MDG structure corresponding to the loop body.

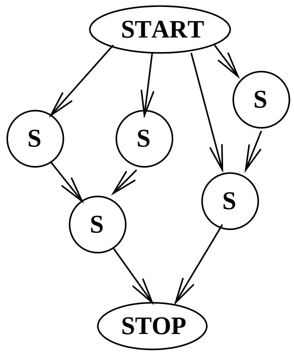


Figure 3.1 Example of a Simple MDG

- Conditional (`C`) nodes that correspond to conditional constructs in the given application. A `C` node contains two MDG structures corresponding to the `THEN` and `ELSE` bodies.

- User-Defined (`U`) nodes that correspond to user-defined functions in the given application. A `U` node contains an MDG corresponding to the function body.

Edges in the MDG correspond to precedence constraints that exist between tasks. To understand the MDG structure more clearly, we have shown examples of MDGs in Figures 3.1 and 3.2. Figure 3.1 illustrates an MDG with only `S` nodes; we refer to such MDGs as *Simple* MDGs. Figure 3.2 depicts an MDG with `L`, `C` and `U` nodes in addition to `S` nodes; we refer to such MDGs as *Hierarchical* MDGs. The automatic extraction of the MDG structure from Fortran and MATLAB programs is discussed in Section 3.2.

The weights of nodes and edges in the MDG are based on the concepts of *Processing* and *Data transfer* costs. Processing costs account for the communication and computation costs of data parallel tasks corresponding to nodes and depend on the number of processors allocated to the node. Data transfer costs account for the costs of any data redistribution that is necessary for preserving data dependence relationships between nodes. Section 3.3 describes the processing and data transfer costs in detail and develops mathematical models for them.

3.2 MDG Extraction

Sections 3.2.1 and 3.2.2 consider the extensions required for HPF and MATLAB in order to facilitate the extraction of an MDG structure. Later, we present algorithms that perform the

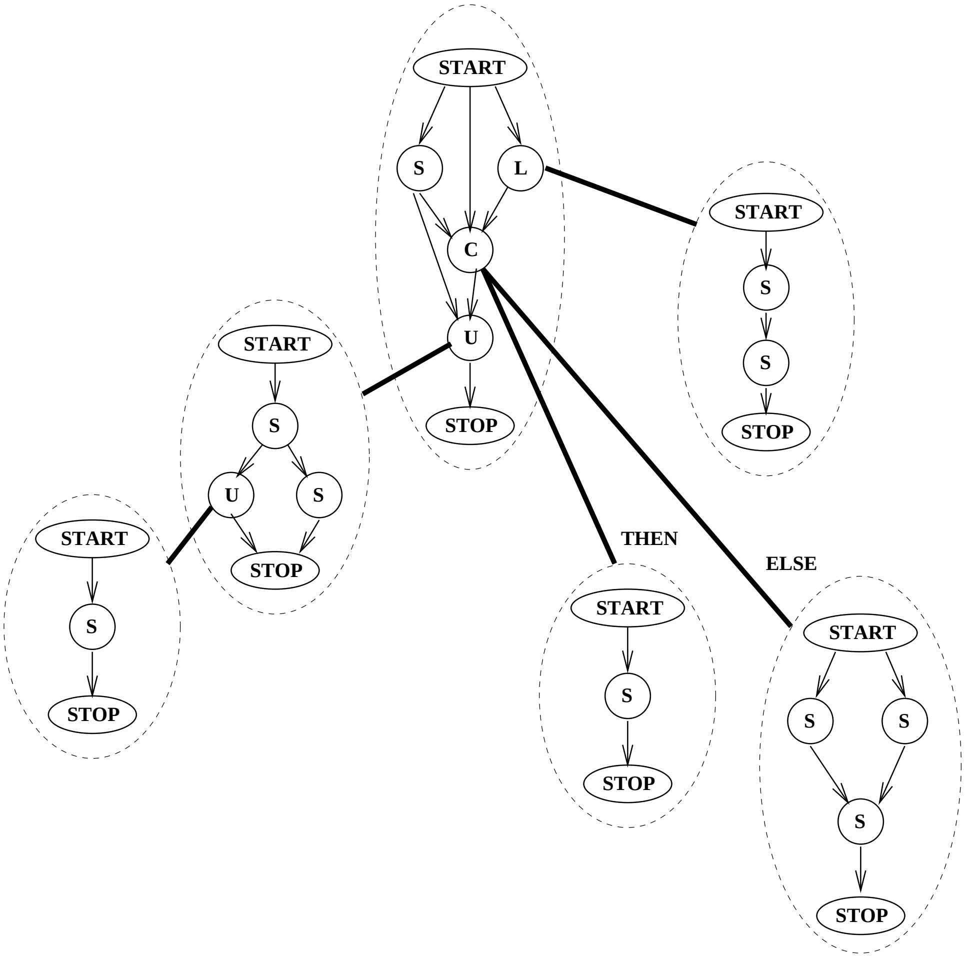


Figure 3.2 Example of a Hierarchical MDG

automatic extraction of the MDG structure in Section 3.2.3. Section 3.3.5 presents algorithms used to fill in cost information in the MDG.

3.2.1 Extensions to High Performance Fortran

In order to let users specify an MDG structure in the HPF framework, we provide the `PROCESSES` directive:

```
CHPF$   PROCESSES
        statements
CHPF$   ENDPROCESSES
```

The `PROCESSES` directive is used to demarcate an MDG structure in the source program. At this point, we do not allow for nested `PROCESSES` directives. Further, only a single `PROCESSES` directive is allowed per subroutine or function and all statements in the subroutine must be enclosed inside the `PROCESSES` directive. Only the following types of statements are allowed inside the `PROCESSES` directive:

- Calls to intrinsic or user-defined HPF subroutines that are `PURE` [1],[2]. All parameters to these calls must be complete arrays or scalars (array element accesses not allowed). This greatly simplifies our dependence analysis algorithms discussed later.
- Assignment statements involving scalar computations (array element accesses not allowed).
- `DO` loops with unit stride and constant lower and upper bounds.
- `DO WHILE` loops with scalar variables (array element accesses not allowed) in the predicate expression.
- `IF` statements with scalar variables (array element accesses not allowed) in the predicate expression.

A few examples illustrating the use of the processes directive are shown below:

```
CHPF$   PROCESSES
        CALL MATRIXMULTIPLY(A,B,S1)
        CALL MATRIXMULTIPLY(C,D,S2)
        CALL MATRIXADD(S1,S2,S)
CHPF$   ENDPROCESSES

CHPF$   PROCESSES
```

```
        DO I=1,20
           CALL READIMAGE(AR,AI)
           CALL ROWFFT(AR,AI)
           CALL COLFFT(AR,AI)
        END DO
CHPF$   ENDPROCESSES

CHPF$   PROCESSES
        CALL NORM(B,BNRM)
        IF (BNRM.GT.0)
           SOLVE(A,B,X)
        ENDIF
CHPF$   ENDPROCESSES
```

The PROCESSES directive was originally proposed by Foster and Chandy [43]. Their version of the directive allows only subroutine calls in the body. These subroutines are instantiated simultaneously and exchange data between themselves using explicit interfaces called CHANNELs. Communication using CHANNELs results in implicit precedence relationships between subroutines. Foster and Chandy also have a loop directive called the PROCESSDO directive. This allows the user to create multiple instances of the same subroutine.

Our use of the PROCESSES directive is essentially similar to the use of the PARALLEL directive by Gross, O'Halloran and Subhlok [8]. A principal difference is that we allow for WHILE loops and IF statements in addition to the FOR loops allowed in the PARALLEL directive. Just as in the PARALLEL directive, we use explicit dependence analysis to determine precedence relationships between tasks and these precedence relationships are preserved during program execution.

To ease our dependence analysis algorithms, we provide another set of directives for the user. Similar directives are provided by Gross, O'Halloran and Subhlok for this purpose. The directives are

CHPF$ IN *variables*

CHPF$ OUT *variables*

CHPF$ INOUT *variables*

The IN directive declares input variables for a subroutine, the OUT directive declares output variables and the INOUT directive declares variables that are inputs and well as outputs. Input/output directives must be provided for all user-defined subroutines in the program. For intrinsic functions, information about input and output variables is implicit. Examples illustrating the use of the IN, OUT and INOUT directives are shown below:

```
        SUBROUTINE MATRIX_ADD(A,B,C)
CHPF$   IN A,B
```

```
CHPF$   OUT C

        SUBROUTINE DOT_PRODUCT(A,DP)
CHPF$   IN A
CHPF$   OUT DP

        SUBROUTINE FFT2D(AR,AR)
CHPF$   INOUT AR,AI
```

To provide a more complete picture of the use of our directives, we have shown the extended HPF code for one of our benchmark applications in Figure 3.3. The application corresponds to a biconjugate gradient method for solving systems of linear equations [65].

3.2.2 Extensions to MATLAB

In addition to providing extensions to HPF, we also provide a compiler for MATLAB programs. The language supported by our compiler currently is a subset of MATLAB [4] with additional directives for specifying type and size information for variables. These directives are in the form of MATLAB comments to ensure that the code can also be used in the MATLAB computing environment.

A MATLAB program is assumed to be composed of a set of functions or modules along with a `main` function. Program execution is assumed to start with the first statement in `main`. A function has a header, types and size directives for variables used and a set of statements corresponding to the body. Each of these parts of a function is described below.

Header The header provides a declaration for the function name, its input variables and output variables.

`function [`*outputs*`] = `*name*`(`*inputs*`)`

Examples:

```
function [a,b,c] = foo1(x,y)
function a = foo2(x,y,z)
```

Type and size directives These directives provide type and size information for all variables used in the function.

`%!` *category type name*`(`*dimensions*`);`

Examples:

```
      SUBROUTINE BICG(N,A,X,B,M,TOL,Y,ERROR)
CHPF$ IN N,A,X,B,M,TOL
CHPF$ OUT Y,ERROR
      INTEGER N
      REAL A(N,N),M(N,N),X(N),Y(N),TOL,ERROR
      REAL R(N),R_TLD(N),Z(N),Z_TLD(N),P(N)
      REAL P_TLD(N),Q(N),Q_TLD(N),X1(N)
      REAL BNRM2,ALPHA,BETA,RHO,RHO_1,RHO_T
CHPF$ PROCESSES
      CALL NORM(B,BNRM2)
      CALL COPY(Y,X)
      IF (BNRM2.NE.0) THEN
        CALL SOLVE(M,R,Z)
        CALL SOLVE(M,R_TLD,Z_TLD)
        CALL DOTP(Z,R_TLD,RHO)
        IF (RHO.NE.0) THEN
          CALL COPY(Z,P)
          CALL COPY(Z_TLD,P_TLD)
          CALL MATVEC(A,P,Q)
          CALL MATVEC(A,P_TLD,Q_TLD)
          CALL DOTP(P_TLD,Q,RHO_T)
          CALL VECSCALE(P,(RHO/RHO_T),X1)
          CALL PLUS(X,X1,X)
          CALL VECSCALE(Q,(RHO/RHO_T),X1)
          CALL MINUS(R,X1,R)
          CALL VECSCALE(Q_TLD,(RHO/RHO_T),X1)
          CALL MINUS(R_TLD,X1,R_TLD)
          CALL NORM(R,ERROR)
          DO WHILE ((ERROR/BNRM2).GT.TOL)
            CALL SOLVE(M,R,Z)
            CALL SOLVE(M,R_TLD,Z_TLD)
            CALL DOTP(Z,R_TLD,RHO)
            IF (RHO.NE.0) THEN
              CALL VECSCALE(P,(RHO/RHO_1),X1)
              CALL PLUS(Z,X1,P)
              CALL VECSCALE(P_TLD,(RHO/RHO_1),X1)
              CALL PLUS(Z_TLD,X1,P_TLD)
              CALL MATVEC(A,P,Q)
              CALL MATVEC(A,P_TLD,Q_TLD)
              CALL DOTP(P_TLD,Q,RHO_T)
              CALL VECSCALE(P,(RHO/RHO_T),X1)
              CALL PLUS(X,X1,X)
              CALL VECSCALE(Q,(RHO/RHO_T),X1)
              CALL MINUS(R,X1,R)
              CALL VECSCALE(Q_TLD,(RHO/RHO_T),X1)
              CALL MINUS(R_TLD,X1,R_TLD)
              CALL NORM(R,ERROR)
            ENDIF
          ENDDO
        ENDIF
      ENDIF
CHPF$ ENDPROCESSES
      RETURN
```

Figure 3.3 Extended HPF Code for the BiConjugate Gradient Algorithm

```
%! local float foo1(128,128);
%! input int foo2($,$);
%! output double foo3($),foo4;
```

where *local* variables are used only in the body of the function, *input* variables are arguments passed in and *output* variables are modified by the function. Types for arrays include integer (`int`), single precision (`float`) and double precision (`double`). Dimensionality information is required for all variables. However, size information is optional – it is required for variables whose sizes are invocation independent and not required for variables whose sizes are invocation dependent. A `$` for a particular dimension denotes that the size in that dimension is invocation dependent. The compiler automatically determines the sizes of input variables for each invocation of the function.

Statements Currently, we allow for simple assignment statements, simple `FOR` and `WHILE` loops and simple `IF` statements in the body of a function.

A simple assignment statement has the form:

```
[outputs] = expression
```

where the expression can use any of the MATLAB operators or intrinsic functions. For details on MATLAB operators, refer to the MATLAB Reference Guide [4].

Examples:

```
[a,b,c]=foo1(x,y);
a=x*y+z*foo2(u,v,w);
```

At this point, we only allow references to entire arrays in expressions although MATLAB allows for subarray references.

A simple `FOR` loop has the form:

```
for loopvar = start:stop
    statements
end
```

where statements can be any of the types of statements we allow.

Examples:

```
for i=1:100
    a=x*y-z;
    b=foo1(z);
end
```

A simple `WHILE` loop has the form:

```
while (var1 operator var2)
    statements
end
```

where statements can be any of the types of statements we allow. The variables in the predicate have to be scalars.

Examples:

```
while (error<tol)
    solve(a,b,c);
    error=norm(c);
end
```

A simple IF statement has the form:

```
if (var1 operator var2)
    statements
else
    statements
end
```

where statements can be any of the types of statements we allow. The variables in the predicate have to be scalars.

Examples:

```
if (rho~=0)
    vecscale(a,rho,x);
    matvec(b,x,y);
else
    solve(b,a,x);
end
```

To provide a more complete picture of the use of our extensions to MATLAB, we have shown the extended MATLAB code for a biconjugate gradient method of solving a system of linear equations [65] in Figure 3.4.

3.2.3 Extraction of MDG structure

For the purposes of MDG extraction, we treat Extended HPF and Extended MATLAB programs the same by using the same internal parse tree format for both types of applications. For extended HPF applications, we use Parafrase-2 [44],[66] (with modifications to accommodate extended HPF [27]) as a front end to build this parse tree. For extended MATLAB programs, we have built a parser.

As a first step, we traverse the parse tree and build an MDG structure for each program unit (HPF subroutine/MATLAB function). The MDG for a program unit is built in an incremental

```
function [y, error] = bicg(A, x, b, M, tol)
%! input float A(:,:),x(:,:),b(:,:),M(:,:),tol;
%! output float y(:,:),error;
%! local float r(:,:),r_tld(:,:),z(:,:),z_tld(:,:);
%! local float p(:,:),p_tld(:,:),q(:,:),q_tld(:,:);
%! local float bnrm2,alpha,beta,rho,rho_1;
y=x;
bnrm2 = norm(b);
if (brnrm2 ~= 0)
  z = solve(M,r);
  z_tld = solve(M',r_tld);
  rho = dotp(z,r_tld);
  if (rho ~= 0)
    p = z;
    p_tld = z_tld;
    q = matvec(A,p);
    q_tld = matvec(A',p_tld);
    alpha = rho / dotp(p_tld*q);
    y = y + vecscale(p,alpha);
    r = r - vecscale(q,alpha);
    r_tld = r_tld - vecscale(q_tld,alpha);
    error = norm(r) / bnrm2;
    while (error > tol)
      z = solve(M,r);
      z_tld = solve(M',r_tld);
      rho = dotp(z,r_tld);
      if (rho ~= 0)
        beta = rho / rho_1;
        p = z + vecscale(p,beta);
        p_tld = z_tld + vecscale(p_tld,beta);
        q = matvec(A,p);
        q_tld = matvec(A',p_tld);
        alpha = rho / dotp(p_tld*q);
        y = y + vecscale(p,alpha);
        r = r - vecscale(q,alpha);
        r_tld = r_tld - vecscale(q_tld,alpha);
        error = norm(r) / bnrm2;
      end
    end
  end
end
```

Figure 3.4 Extended MATLAB Code for the BiConjugate Gradient Algorithm

```
BuildMDGForStmtList(List)
if List = NIL
  then return NIL
MDG ← BuildMDGForStmt(stmt[List])
List ← next[List]
while List ≠ NIL
      do TempMDG ← BuildMDGForStmt(stmt[List])
         MDG ← JoinMDG(MDG, TempMDG)
         List ← next[List]
return MDG
```

Figure 3.5 Building an MDG for a List of Statements

```
BuildMDGForStmt(Stmt)
switch
  case type[Stmt] = Assignment :
       MDG ← BuildMDGForAsgnStmt(asgnstmt[Stmt])
  case type[Stmt] = Conditional :
       MDG ← BuildMDGForCondStmt(condstmt[Stmt])
  case type[Stmt] = Loop :
       MDG ← BuildMDGForLoopStmt(loopstmt[Stmt])return MDG
```

Figure 3.6 Building an MDG for a Statement

manner by traversing the statement list. This is shown in Figure 3.5. Basically, we build an MDG for each statement and use the JoinMDG procedure to add this MDG to the MDG corresponding to all previous statements. We will discuss the JoinMDG procedure slightly later in this section. In Figure 3.6 we show the procedure for building the MDG for a generic statement. This procedure is straightforward, we simply use the type of statement to select the appropriate routine.

Figure 3.7 shows the procedures used for building the MDG structure for an assignment statement. The BuildMDGForExpr procedure is central to building the MDG for an assignment statement. Each of the function calls and operators in the expression tree for the assignment statement correspond to a Simple `S` node in the MDG. Dependence relationships between function calls and operators in the expression tree correspond to edges in the MDG. BuildMDGForExpr traverses the expression tree in a bottom-up manner – MDG nodes are created for the parameters to a function call or the arguments to an operator before adding a

```
BuildMDGForAsgnStmt(Stmt)
MDG ← New()
BuildMDGForExpr(MDG, expr[Stmt])
return MDG

BuildMDGForExpr(MDG, Expr)
if Expr = nil or type[Expr] = VarRef
   then return MDGNode ← New()
if type[Expr] = FuncCall
   then type[MDGNode] ← type[func[Expr]]
        func[MDGNode] ← func[Expr]

   else  type[MDGNode] ← OperatorNode
        expr[MDGNode] ← Expr

ExprList ← args[Expr]
while ExprList ≠ nil
      do BuildMDGForExpr(expr[ExprList])
         if type[expr[ExprList]] = VarRef
            then Var ← var[expr[ExprList]]
                 AddEdge(startnode[MDG], MDGNode, Var)
            else  Var ← outvar[expr[ExprList]]
                 AddEdge(mdgnode[expr[ExprList]], MDGNode, Var)
         ExprList ← next[ExprList]
if parent[Expr] = nil
   then VarList ← lhs[stmt[Expr]]
        while VarList ≠ nil
              do AddEdge(MDGNode, stopnode[MDG], var[VarList])
                 VarList ← next[VarList]
   else  Var ← CreateTempOutVar(Expr)
        outvar[Expr] ← Var
mdgnode[Expr] ← MDGNode
```

Figure 3.7 Building an MDG for an Assignment Statement

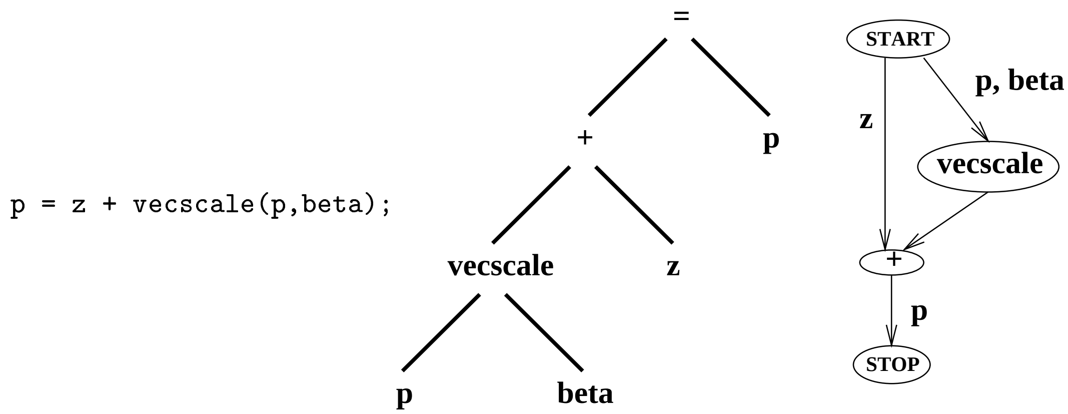


Figure 3.8 Example of MDG Construction for an Assignment Statement

node for the function call or operator itself. This is illustrated in Figure 3.8. The MATLAB assignment statement shown has a function call and an operator. The expression tree and the corresponding MDG structure are shown alongside. If a function call parameter or operator argument is a variable reference, we add an edge between the `START` node of the MDG and the `S` node corresponding to the function call or operator. Similarly, if a function call parameter or operator output results in a variable write, we add an edge between the `S` node corresponding to the function call or operator and the `STOP` node of the MDG. In the example of Figure 3.8, the variables referenced are `z`, `p` and `beta`. The variable written is `p`. Building the MDG in this manner provides us with a summary of variable reference and write information in the `START` and `STOP` nodes, respectively. As we will see during the discussion of the JoinMDG routine that follows, this summary is very useful.

The JoinMDG procedure is shown in Figure 3.9. This procedure combines the information in a given pair of MDGs to construct a single MDG. In order to do this, we consider the list of variables written by the first MDG and compare it against the list of variables referenced in the second MDG. As discussed above, these lists are readily provided by the `STOP` node of the first MDG and the `START` node of the second MDG respectively. If we find a variable referenced in the second MDG is written in the first MDG, we look for the node that last modifies the variable in the first MDG and connect all nodes that reference the variable in the second MDG to that node. If a variable is written in the first MDG and not written in the second MDG, the last node to write to the variable is connected to the `STOP` node of the second MDG. Similarly, if a variable is referenced in the second MDG but not written in the first MDG, we connect all nodes that reference the variable in the second MDG to the `START` node

```
JoinMDG(MDG1, MDG2)
VarList1 ← inlist[stopnode[MDG1]]
while VarList1 ≠ NIL
      do VarList2 ← outlist[startnode[MDG2]]
         while (VarList2 ≠ NIL) or (name[var[VarList1]] ≠ name[var[VarList2]])
               do VarList2 ← next[VarList2]

         if VarList2 ≠ NIL
           then MDGNode1 ← GetParent(stopnode[MDG1], var[VarList1])
                MDGNode2 ← GetChild(startnode[MDG2], var[VarList2])
                while MDGNode2 ≠ NIL
                      do AddEdge(MDGNode1, MDGNode2, var[VarList1])
                         MDGNode2 ← next[MDGNode2]

         VarList2 ← inlist[stopnode[MDG2]]
         while (VarList2 ≠ NIL) or (name[var[VarList1]] ≠ name[var[VarList2]])
               do VarList2 ← next[VarList2]

         if VarList2 = NIL
           then AddEdge(MDGNode1, stopnode[MDG2], Var1)
         VarList1 ← next[VarList1]VarList2 ← outlist[startnode[MDG2]]
while VarList2 ≠ NIL
      do VarList1 ← inlist[stopnode[MDG1]]
         while (VarList1 ≠ NIL) or (name[var[VarList1]] ≠ name[var[VarList2]])
               do VarList1 ← next[VarList1]

         if VarList2 = NIL
           then MDGNode2 ← GetChild(startnode[MDG2], var[VarList2])
                while MDGNode2 ≠ NIL
                      do AddEdge(startnode[MDG1], MDGNode2, var[VarList2])
                         MDGNode2 ← next[MDGNode2]

         VarList2 ← next[VarList2]
FreeMDGNode(stopnode[MDG1])
FreeMDGNode(startnode[MDG2])
```

Figure 3.9

```
p = z + vecscale(p,beta);
q = matvec(A,p);
```

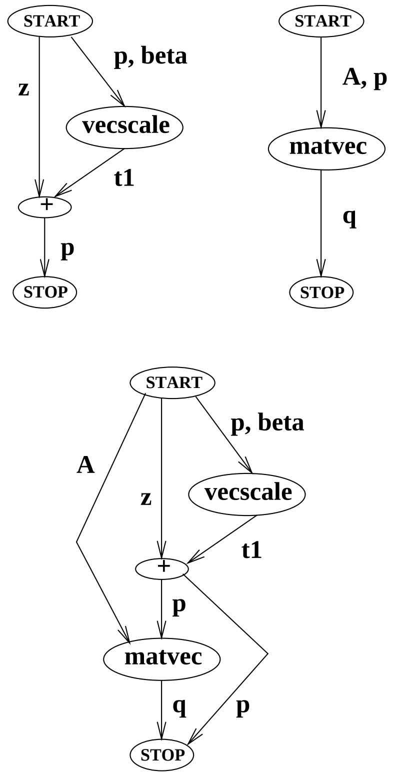


Figure 3.10 Example of MDG Combining

of the first MDG. Lastly, we free the `STOP` node of the first MDG and the `START` node of the second MDG. Figure 3.10 illustrates the combining of two MDGs corresponding to consecutive statements in the program. The first MDG references `p`, `z`, and `beta` and writes to `p`. The second MDG references `A` and `p` and writes to `q`. Following the JoinMDG procedure, we see that `p` is a variable written in the first MDG and referenced in the second MDG, so we connect the output of the `+` node to the `matvec` node. `p` is also a variable that is written in the first MDG but not written in the second MDG, so we connect the `+` node to the `STOP` node of the second MDG. Finally, `A` is referenced in the second MDG but not written in the first MDG, so we connect the `matvec` node to the `START` node of the first MDG.

Figure 3.11 shows the procedure used for building the MDG structure for a conditional statement. This procedure is fairly straightforward and results in a hierarchical MDG. At the top level, we build an MDG with a single Conditional `C` node. All variables referenced in the predicate result in edges from the `START` node of the MDG to the `C` node. We then build separate MDGs at a lower level for the statements in the `THEN` and the `ELSE` parts. Finally, we obtain all variable reference and write information from these MDGs (using their `START` and

```
BuildMDGForCondStmt(Stmt)
MDG ← New()
MDGNode ← New()
type[MDGNode] ← ConditionalNode
predicate[MDGNode] ← predicate[Stmt]
VarList ← GetVarRefsInExpr(predicate[MDGNode])
while VarList ≠ nil
        do AddEdge(startnode[MDG], MDGNode, var[VarList]
            VarList ← next[VarList]
ifmdg[MDGNode] ← BuildMDGForStmtList(thenbody[Stmt])
VarList ← outlist[startnode[thenmdg[MDGNode]]]
while VarList ≠ nil
        do AddEdge(startnode[MDG], MDGNode, var[VarList])
            VarList ← next[VarList]
VarList ← inlist[stopnode[thenmdg[MDGNode]]]
while VarList ≠ nil
        do AddEdge(MDGNode, stopnode[MDG], var[VarList])
            VarList ← next[VarList]
elsemdg[MDGNode] ← BuildMDGForStmtList(elsebody[Stmt])
VarList ← outlist[startnode[elsemdg[MDGNode]]]
while VarList ≠ nil
        do AddEdge(startnode[MDG], MDGNode, var[VarList])
            VarList ← next[VarList]
VarList ← inlist[stopnode[elsemdg[MDGNode]]]
while VarList ≠ nil
        do AddEdge(MDGNode, stopnode[MDG], var[VarList])
            VarList ← next[VarList]
return MDG
```

Figure 3.11 Building an MDG for a Conditional Statement

```
if (rho~=0)
  p = solve(A,z);
  q = solve(A,p);
else
  p = z + vecscale(p,beta);
  q = matvec(A,p);
endif
```

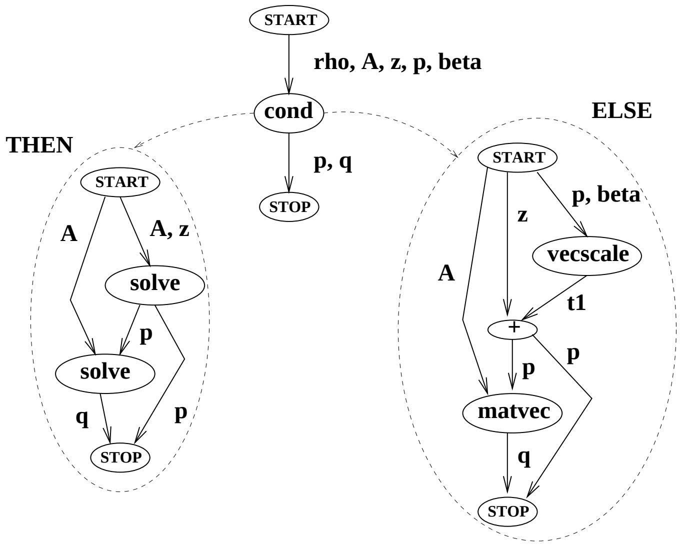


Figure 3.12 Example of MDG Construction for a Conditional Statement

STOP nodes) and use this to update the variable reference and write information at the top level. All variable references in the lower levels result in edges between the START node and the C node at the top level. Similarly, variable writes result in edges between the C node and the STOP node at the top level. Figure 3.12 illustrates an example of a conditional statement and the MDG corresponding to it. At the top level, we create an MDG with a single C node and add an edge from START to the C node for the reference to rho in the predicate. We then build the MDGs corresponding to the THEN and ELSE parts using the procedures described earlier. The THEN part references A, z, p and beta; it writes p and q. On the other hand, the ELSE part references A and z; it writes p and q. This variable reference and write information is now used to add variables to edges at the upper level. Variables A, z, p and beta are added to the edge between START and the C node; variables p and q are added to the edge between the C node and STOP.

Figure 3.13 shows the procedure for building the MDG structure for a loop statement. Again, this procedure results in a hierarchical MDG. At the top level, we build an MDG with a single Loop L node. If the loop is a WHILE loop, we add edges between the START node and the L node for variable references in the predicate. We then build an MDG at a lower level for the statements in the loop body. Finally, just as in the case of a conditional statement, we obtain variable reference and write information to update variable reference and write information at the top level. Figure 3.14 illustrates an example of a loop statement and the MDG corresponding

```
BuildMDGForLoopStmt(Stmt)
MDG ← New()
MDGNode ← New()
type[MDGNode] ← LoopNode
if type[Stmt] = While
   then predicate[MDGNode] ← predicate[Stmt]
        VarList ← GetVarRefsInExpr(predicate[MDGNode])
        while VarList ≠ nil
              do AddEdge(startnode[MDG], MDGNode, var[VarList]
                 VarList ← next[VarList]
loopmdg[MDGNode] ← BuildMDGForStmtList(loopbody[Stmt])
VarList ← outlist[startnode[loopmdg[MDGNode]]]
while VarList ≠ nil
      do AddEdge(startnode[MDG], MDGNode, var[VarList])
         VarList ← next[VarList]
VarList ← inlist[stopnode[loopmdg[MDGNode]]]
while VarList ≠ nil
      do AddEdge(MDGNode, stopnode[MDG], var[VarList])
         VarList ← next[VarList]
return MDG
```

Figure 3.13 Building an MDG for a Loop Statement

```
for i=1:10
  p = solve(A,z);
  q = matvec(A,p);
  rho = norm(p);
  r = vecscale(q,rho);
end
```

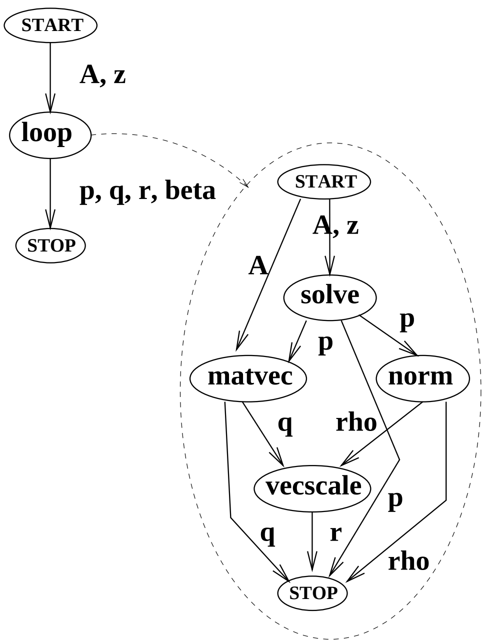


Figure 3.14 Example of MDG Construction for a Loop Statement

to it. At the top level, we create an MDG with a single `L` node. Since the loop is a `FOR` loop, there are no variables referenced. We now build an MDG corresponding to the loop body. The variables referenced in this MDG are `A` and `z`; variables written are `p`, `q`, `r` and `beta`. This information is used to add variables to edges at the upper level. `A` and `z` are added to the edge between `START` and the `L` node; `p`, `q`, `r` and `beta` are added to the edge between the `L` node and `STOP`.

3.3 Mathematical Cost Models

This section deals with the important aspect of choosing appropriate functions to represent the processing and data transfer costs involved in an MDG. The cost functions we choose have to be accurate in practice and belong to the class of posynomial functions (Appendix A). The need for the latter property arises from our allocation and scheduling algorithms described in Chapter 4. In this section we describe our processing and data transfer cost functions and prove that they all belong to the class of posynomial functions. The processing cost function we use is an often used model. On the other hand, we have developed our own data transfer cost functions.

3.3.1 Processing cost model

For the processing cost model, we use Amdahl's law, i.e., the execution time of the data parallel task corresponding to the ith node (t_i^C) as a function of the number of processors it uses (p_i) is given by

$$t_i^C = (\alpha_i + \frac{1 - \alpha_i}{p_i}) \cdot \tau_i \tag{3.1}$$

where τ_i is the execution time of the task on a single processor and α_i is the fraction of the data parallel task that has to be executed serially. It can be seen that

$$\begin{aligned} 0 \leq \alpha_i \leq 1 \\ 0 \leq \tau_i \end{aligned} \tag{3.2}$$

The values of parameters α and τ are calculated for the data parallel tasks used in our benchmarks by actually measuring execution times for the tasks as a function of the number of processors they use and then using linear regression to fit the measured values to a function of the form shown above.

Lemma 3.1 *t_i^C is a posynomial function w.r.t. p_i.*

Proof : From Equation (3.1) we see that t_i^C is made of two components – a constant component $\alpha_i \cdot \tau_i$ and a variable component $\frac{(1-\alpha_i)\cdot\tau_i}{p_i}$. The first component is a posynomial since it is a nonnegative constant (under Equation (3.2)). The second component has a nonnegative constant factor multiplying a posynomial $\frac{1}{p_i}$. Under the constant property discussed in Appendix A, this component is also a posynomial. Since both components are posynomials, using the sum property of Appendix A, t_i^C is a posynomial. □

Lemma 3.2 *$t_i^C \cdot p_i$ is a posynomial function w.r.t. p_i.*

Proof : Using Equation (3.1), we obtain

$$t_i^C \cdot p_i = \alpha_i \cdot \tau_i \cdot p_i + (1 - \alpha_i) \cdot \tau_i \tag{3.3}$$

This equation has two components – a variable component $\alpha_i \cdot \tau_i \cdot p_i$ which has a nonnegative constant factor $\alpha_i \cdot \tau_i$ multiplying a posynomial p_i. By the constant property of Appendix A, this component is a posynomial. The other component in the equation above is a nonnegative constant which is also a posynomial. Hence, using the sum property of Appendix A, we see that $t_i^C \cdot p_i$ is a posynomial. □

It must be pointed out that α and τ for a task could depend on the size of data input to the task. This does not affect the statements made in either Lemma above.

3.3.2 Data transfer cost model

In this section we consider the cost of redistribution of an array of data elements between the execution of nodes i and j of the MDG. Node i is assumed to have p_i processors allocated to it and node j is assumed to have p_j processors allocated to it. For modeling an array redistribution, we assume that the array is distributed evenly across the p_i sending processors before the redistribution and across the p_j receiving processors after the redistribution. In addition, we assume that the sizes and numbers of messages will be the same for each sending processor and for each receiving processor. For example, every sending processor may send three messages of 1000 bytes each and every receiver may receive five messages of 1500 bytes each. Both of our assumptions are valid for the realm of regular scientific applications.

The regular distributions of an array along any of its dimensions (size along dimension is S) across a set of p processors are classified into the following cases

- `ALL` : All elements of the array along the dimension are owned by the same processor ($p = 1$)

- `BLOCK` : Elements of the array are distributed evenly across all the processors with each processor owning a contiguous block of $\frac{S}{p}$ elements.

- `CYCLIC` : Elements of the array are distributed evenly across all the processors in a round robin fashion with each processor owning every pth element with the ith processor starting at element i.

- `BLOCKCYCLIC(X)` : Elements of the array are distributed evenly across all the processors in a round robin fashion with each processor owning every pth block of X elements, the ith processor starting at the ith block of X elements.

Figure 3.15 illustrates a few different regular distributions of an array. More details of regular distributions can be found in [1],[2]. For our discussion of data transfer costs, the distribution of an array can change from any of those listed above to any other along one or more of its dimensions.

For an array redistribution from node i to node j, there are three basic cost components – a sending component t^S_{ij} incurred by processors allocated to node i, a network delay component t^D_{ij} to allow for messages to reach the processors allocated to node j, and a receiving component t^R_{ij} incurred by processors allocated to node j. The t^S_{ij} component is accounted for in the weight of node i, the t^D_{ij} component is taken to be the weight of the edge joining node i and node j, and the t^R_{ij} component is accounted for in the weight of node j.

We propose the following expressions for the three cost components

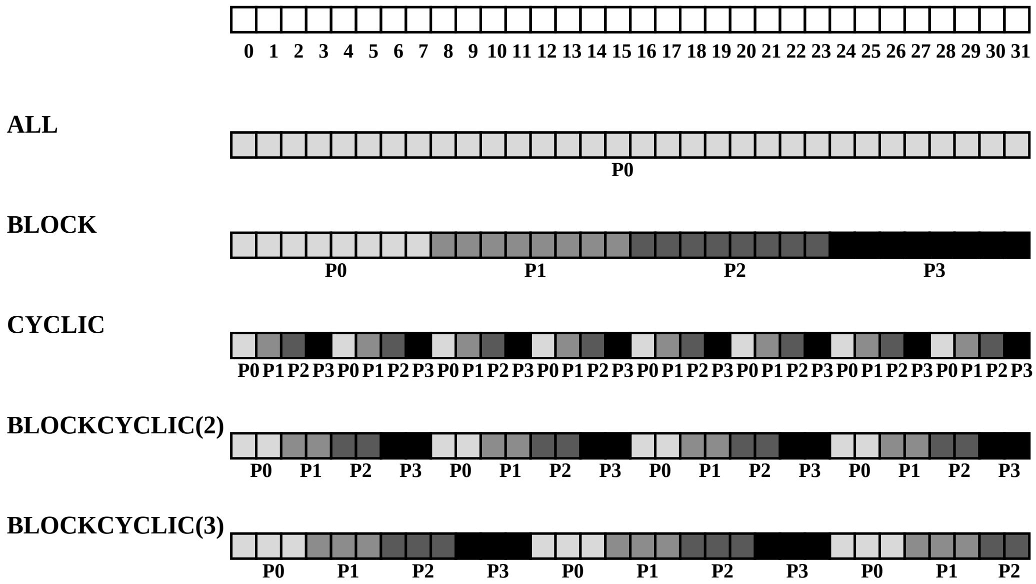


Figure 3.15 Examples of Regular Distributions

$$\begin{aligned} t_{ij}^{S} &= S_{ij}(p_i, p_j) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\ t_{ij}^{D} &= \frac{L}{p_i \cdot S_{ij}(p_i, p_j)} \cdot t_n \\ t_{ij}^{R} &= R_{ij}(p_i, p_j) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr} \end{aligned} \tag{3.4}$$

where

- L is the length (in bytes) of the array being transferred
- t_{ss}, t_{ps} are the startup and per byte cost for sending messages from a processor
- t_n is the network cost per message byte
- t_{sr}, t_{pr} are the startup and per byte cost for receiving messages at a processor
- S_{ij} is the number of messages sent from each sending processor
- R_{ij} is the number of messages received at each receiving processor.

Intuitively, the sending component (t_{ij}^{S}) for each sending processor involves a startup cost for each of the S_{ij} messages sent and a processing cost for its share of the array ($\frac{L}{p_i}$). The same

Table 3.1 Block Factors for Various Regular Distributions

Distribution	Block Factor
`ALL`	L
`BLOCK`	$\frac{L}{p_i}$
`CYCLIC`	1
`BLOCKCYCLIC(X)`	X

Table 3.2 Skip Factors for Various Regular Distributions

Distribution	Skip Factor
`ALL`	L
`BLOCK`	L
`CYCLIC`	p_i
`BLOCKCYCLIC(X)`	$X \cdot p_i$

logic holds for the receiving component (t_{ij}^R) of receiving processors. The network component represents the minimum delay required for messages to be delivered to the receiving processors after they have been sent from the sending processors. If we assume a pipelined network with no congestion effects, this delay will depend on the length of the last message sent. By our assumption of equal-sized messages, we see that the size of each message will be $\frac{L}{p_i \cdot S_{ij}(p_i, p_j)}$. This is the reasoning behind the network cost component expression shown.

The quantities S_{ij} and R_{ij} in our cost expressions will depend on the type of array redistribution being performed. It is possible to express these quantities in terms of a pair of parameters of the sending and receiving distributions. The first of these parameters is called the *Block Factor* (BF) and it provides a measure of the sizes of the blocks of elements a processor owns under any of the regular distributions. The block factor for the different regular distributions of an array of L bytes on p_i processors is shown in Table 3.1. The other parameter we use is called the *Skip Factor* (SF) which provides an idea of the distance between the successive blocks of elements a processor owns. We have listed the skip factors for the various regular distributions of an array of L bytes on p_i processors in Table 3.2. The quantities S_{ij} and R_{ij} can be expressed in terms of block factors and skip factors using

$$
\begin{aligned}
S_{ij} &= \max(1, \frac{SF_j}{SF_i}, \frac{BF_i}{BF_j}, \frac{SF_j}{SF_i} \cdot \frac{BF_i}{BF_j}) \\
R_{ij} &= \max(1, \frac{SF_i}{SF_j}, \frac{BF_j}{BF_i}, \frac{SF_i}{SF_j} \cdot \frac{BF_j}{BF_i})
\end{aligned}
\tag{3.5}
$$

where BF_i and SF_i are the block factor and skip factor for the sending distribution; BF_j and SF_j are the block factor and skip factor for the receiving distribution.

Some details have been omitted in the expressions above to make them easily understandable. First, we have considered the redistribution of a one-dimensional array; however, in practice, arbitrary n-dimensional arrays may be redistributed. In addition, we consider redistribution of a single array; again, in practice, more than one array may have to be redistributed between a pair of nodes with the type of redistribution being different for each of the arrays. It is easy to extend our functions to account for these effects. We do not show these extended forms here as they are complex and lengthy. Our actual implementation of the cost models uses extended forms of the functions described above.

Lemma 3.3 *t^S_{ij}, t^R_{ij} and t^D_{ij} are posynomial functions w.r.t. p_i and p_j for all possible cases of redistributions.*

Proof : A complete proof would require us to show that the statement above is true for all cases of redistributions. The lack of space prevents us from presenting proofs for all cases. We show here that the statement holds for two of the cases, similar proofs can be derived for all the other cases.

- Case A: `BLOCK` to `BLOCK`.

  For this case, the expressions for S_{ij} and R_{ij} (using Tables 3.1 and 3.2 and Equation (3.5)) are given by

$$
\begin{aligned}
S_{ij} &= \max(1, \frac{p_j}{p_i}) \\
R_{ij} &= \max(1, \frac{p_i}{p_j})
\end{aligned}
\tag{3.6}
$$

Using these values in Equation (3.4), we obtain

$$
\begin{aligned}
t_{ij}^S &= \max(1, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\
t_{ij}^D = \frac{L}{p_i \cdot \max(1, \frac{p_j}{p_i})} \cdot t_n \Rightarrow t_{ij}^D &= \min(\frac{L}{p_i}, \frac{L}{p_j}) \cdot t_n \\
t_{ij}^R &= \max(1, \frac{p_i}{p_j}) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr}
\end{aligned}
\tag{3.7}
$$

Proceeding in a manner similar to the one used for the processing cost function (using the posynomial function examples and properties of posynomial functions), it can easily be shown that t_{ij}^S, t_{ij}^D, and t_{ij}^R are all posynomial functions w.r.t. p_i and p_j.

- Case B: `BLOCKCYCLIC(X)` to `BLOCKCYCLIC(Y)`

  For this case, the expressions for S_{ij} and R_{ij} (using Tables 3.1 and 3.2 and Equation (3.5)) are given by

$$
\begin{aligned}
S_{ij} &= \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i}) \\
R_{ij} &= \max(1, \frac{p_i \cdot Y}{p_j \cdot X}, \frac{Y}{X}, \frac{p_i}{p_j})
\end{aligned}
\tag{3.8}
$$

  Using these values in Equation (3.4), we obtain

$$
\begin{aligned}
t_{ij}^S &= \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\
t_{ij}^D = \frac{L}{p_i \cdot \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i})} \cdot t_n \Rightarrow t_{ij}^D &= \min(\frac{L}{p_i}, \frac{L \cdot Y}{p_j \cdot X}, \frac{L \cdot Y}{p_i}, \frac{L}{p_j}) \cdot t_n \\
t_{ij}^R &= \max(1, \frac{p_i \cdot Y}{p_j \cdot X}, \frac{Y}{X}, \frac{p_i}{p_j}) \cdot t_{ss} + L \cdot \frac{1}{p_j} \cdot t_{ps}
\end{aligned}
\tag{3.9}
$$

  Proceeding in a manner similar to the one used for the processing cost function (using the posynomial function examples and properties of posynomial functions), it can easily be shown that t_{ij}^S, t_{ij}^D, and t_{ij}^R are all posynomial functions w.r.t. p_i and p_j.

Lemma 3.4 *$t_{ij}^S \cdot p_i$ and $t_{ij}^R \cdot p_j$ are posynomial functions w.r.t. p_i and p_j for all possible cases of redistributions.*

Proof : A complete proof would require us to show that the statement above is true for all cases of redistributions. The lack of space prevents us from presenting proofs for all cases. We show here that the statement holds for two of the cases, similar proofs can be derived for all the other cases.

- Case A: `BLOCK` to `BLOCK`.

  As shown in the previous lemma, the expressions for t_{ij}^S and t_{ij}^R are given by

$$\begin{aligned} t_{ij}^S &= \max(1, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\ t_{ij}^R &= \max(1, \frac{p_i}{p_j}) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr} \end{aligned} \tag{3.10}$$

  We can now write the expressions for $t_{ij}^S \cdot p_i$ and $t_{ij}^R \cdot p_j$ as

$$\begin{aligned} t_{ij}^S \cdot p_i &= \max(p_i, p_j) \cdot t_{ss} + L \cdot t_{ps} \\ t_{ij}^R \cdot p_j &= \max(p_j, p_i) \cdot t_{sr} + L \cdot t_{pr} \end{aligned} \tag{3.11}$$

  Proceeding in a manner similar to the one used for the processing cost function (using the posynomial function examples and properties of posynomial functions), it can easily be shown that $t_{ij}^S \cdot p_i$ and $t_{ij}^R \cdot p_j$ are both posynomial functions w.r.t. p_i and p_j.

- Case B: `BLOCKCYCLIC(X)` to `BLOCKCYCLIC(Y)`

  As shown in the previous lemma, the expressions for t_{ij}^S and t_{ij}^R are given by

$$\begin{aligned} t_{ij}^S &= \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\ t_{ij}^R &= \max(1, \frac{p_i \cdot Y}{p_j \cdot X}, \frac{Y}{X}, \frac{p_i}{p_j}) \cdot t_{ss} + L \cdot \frac{1}{p_j} \cdot t_{ps} \end{aligned} \tag{3.12}$$

We can now write the expressions for $t_{ij}^S \cdot p_i$ and $t_{ij}^R \cdot p_j$ as

$$\begin{aligned} t_{ij}^S \cdot p_i &= \max(p_i, \frac{p_j \cdot X}{Y}, \frac{X \cdot p_i}{Y}, p_j) \cdot t_{ss} + L \cdot t_{ps} \\ t_{ij}^R \cdot p_j &= \max(p_j, \frac{p_i \cdot Y}{X}, \frac{Y \cdot p_j}{X}, p_i) \cdot t_{ss} + L \cdot t_{ps} \end{aligned} \tag{3.13}$$

Proceeding in a manner similar to the one used for the processing cost function (using the posynomial function examples and properties of posynomial functions), it can easily be shown that $t_{ij}^S \cdot p_i$, and $t_{ij}^R \cdot p_j$ are both posynomial functions w.r.t. p_i and p_j.

Having shown the statement of Lemma 3.3 true for the two example cases, we extend this result to cover all the possible cases of redistribution. □

3.3.3 Node and edge costs

The cost of a node in the MDG comprises the processing cost for the task to which it corresponds, the receiving cost component (t^R) for all data transfers from predecessor nodes and the sending cost component (t^S) for all data transfers to successor nodes. Formally, the cost of the ith node in the MDG (T_i) is given by

$$T_i = \sum_{p \in PRED_i} t_{pi}^R + t_i^C + \sum_{s \in SUCC_i} t_{is}^S \tag{3.14}$$

where $PRED_i$ is the set of predecessors of node i and $SUCC_i$ is the set of successors of node i.

The cost of the edge between nodes i and j in the MDG (E_{IJ}) is the delay component of the data transfer between the nodes and is given by

$$E_{ij} = t_{ij}^N \tag{3.15}$$

Using the sum property of posynomials described in Appendix A, we can show the node and edge costs to be posynomials.

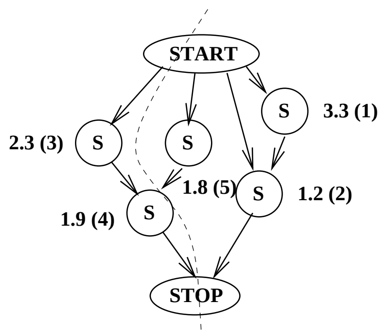


Figure 3.16 Example Illustrating Computation of Critical Path and Average Area

3.3.4 MDG cost properties

We use a few cost-related properties of an MDG in constructing our allocation and scheduling algorithms – the *Critical Path* and the *Average Area*. The critical path (C) of an MDG is defined as the time at which node n (STOP node) finishes execution. If y_i is the finish time of node i, the critical path is given by

$$C = y_n$$

$$y_i = \max_{p \in PRED_i} (y_p + E_p i) + T_i \tag{3.16}$$

The average area (A) of an MDG for a P processor system is defined as

$$A = \frac{1}{P} \cdot \sum_{1}^{n} T_i \cdot p_i \tag{3.17}$$

The critical path represents the longest path in the MDG and the average area provides a measure of the processor-time area required by the MDG. Figure 3.16 shows an example of an MDG with node and edge costs listed. The allocation for each node is shown next to it in parentheses. The critical path and average area (for $P = 8$) are also computed and shown below the MDG.

Using the sum and max properties of posynomials, we can show the critical path and average area to be posynomials. As we shall see in Chapter 4, the fact that the critical path and average area are posynomials is useful while making allocation and scheduling decisions.

3.3.5 Updating MDG cost parameters

The cost models for processing and data transfer in the MDG that were discussed above are all parameterized. In the case of processing costs, the α and τ parameters are characteristic of the data parallel task being considered. For data transfer costs, we essentially need the size of the arrays being redistributed, their distribution for the source task, and their distribution for the target task.

For the purposes of our research, we assume that values of the α and τ parameters and data distribution schemes for arrays are available for all simple (`S`) nodes in the MDG for a given MATLAB or HPF program. In MATLAB programs, tasks corresponding to `S` nodes will be intrinsic function calls or operators. For our research, we also assume that each intrinsic function call or operator in MATLAB maps to a ScaLAPACK routine. We then use profiling information for ScaLAPACK routines to obtain processing cost parameters α and τ as well as the required data distribution schemes. For HPF programs, the `S` nodes correspond to intrinsic calls, operators or calls to user-defined HPF data parallel routines. For intrinsic calls and operators, we use the same approach for filling cost parameters as in MATLAB. In the case of user-defined HPF data parallel routines, cost parameters α and τ are computed using the cost estimation techniques of Gupta and Banerjee [20],[67]. These methods have been implemented as part of the PARADIGM compiler framework [3]. Data distribution information for user-defined HPF routines is obtained from the HPF data distribution directives used in the routine.

Having looked at cost information for `S` nodes that are at the lowest level in the hierarchical MDG structure, we now discuss routines used to fill in cost information at the higher levels in a hierarchical MDG. These routines are shown in Figures 3.17, 3.18, 3.19, and 3.20. Figure 3.17 shows the procedure for filling in cost information for an MDG. We simply traverse the list of nodes in the MDG and invoke the appropriate cost-updating routine for each node. The UPDATECOSTFORSIMPLENODE routine is constructed as discussed in the preceding paragraph. We now discuss the details of the remaining routines that are used for updating costs of compound nodes - loop (`L`) nodes, conditional (`C`) nodes and user-defined (`U`) nodes. To compute the

```
UpdateCostsForMDG(MDG)
  MDGNodeList ← nodes[MDG]
  while MDGNodeList ≠ NIL
        do MDGNode ← node[MDGNodeList]
           switch
              case type[MDGNode] = SimpleNode :
                   UpdateCostForSimpleNode(MDGNode)
              case type[MDGNode] = LoopNode :
                   UpdateCostForLoopNode(MDGNode)
              case type[MDGNode] = ConditionalNode :
                   UpdateCostForConditionalNode(MDGNode)
              case type[MDGNode] = UserDefNode :
                   UpdateCostForUserDefNode(MDGNode)
           MDGNodeList ← next[MDGNodeList]
```

Figure 3.17 Filling in MDG Cost Parameters

```
UpdateCostsForLoopNode(MDGNode)
  UpdateCostsForMDG(loopmdg[MDGNode])
  if looptype[MDGNode] = While
    then Iters ← estiters[MDGNode]for i ← 0 to log2(NumProcs)
          do X[i] ← 2^i
             Y[i] ← Iters · HSAS(loopmdg[MDGNode], X[i])

    else  Iters ← upperbnd[MDGNode] − lowbnd[MDGNode] + 1for i ← 0 to log2(NumProcs)
          do X[i] ← 2^i
             Y[i] ← Iters · HSASForLoop(loopmdg[MDGNode], X[i])

  LinearRegression(X, Y, i, α, τ)
  alpha[MDGNode] ← α
  tau[MDGNode] ← τ

```

Figure 3.18 Filling in Loop Node Cost Parameters

cost parameters for compound nodes, we use the allocation and scheduling algorithms (HSAS and HSASForLoop) discussed in Section 4.3 of Chapter 4.

Figure 3.18 illustrates the routine used for computing the cost of an `L` node. We first update costs for the MDG corresponding to the loop body. Next, we use allocation and scheduling to determine the estimated execution time for the loop body for a range of processor system sizes ranging from 1 to P, where P is the size of the system on which we want to execute the given

```
UpdateCostsForCondNode(MDGNode)
UpdateCostsForMDG(ifmdg[MDGNode])
for i ← 0 to log_2(NumProcs)
do X[i] ← 2^i
    Y[i] ← HSAS(ifmdg[MDGNode], X[i])
LinearRegression(X, Y, i, α_if, τ_if)
UpdateCostsForMDG(elsemdg[MDGNode])
for i ← 0 to log_2(NumProcs)
do X[i] ← 2^i
    Y[i] ← HSAS(elsemdg[MDGNode], X[i])
LinearRegression(X, Y, i, α_else, τ_else)
alpha[MDGNode] ← estif[MDGNode] · α_if + (1 − estif[MDGNode]) · α_else
tau[MDGNode] ← estif[MDGNode] · τ_if + (1 − estif[MDGNode]) · τ_else
```

Figure 3.19 Filling in Conditional Node Cost Parameters

application program. The estimated execution time is scaled by the number of iterations of the loop. In case of `While` loops, we need an estimate that can be obtained through profiling. In the case of `For` loops, we use the lower and upper bounds on the iteration variable. Once we have computed the estimated execution times, we use a LinearRegression routine to obtain our cost model parameters α and τ. The LinearRegression routine computes the values of α and τ that provide the best fit for the estimated execution times under our cost model. We use these values of α and τ for the `L` node. With regard to data distribution schemes for variables, we assume that all arrays required by the loop body are initially distributed on all of the processors allocated to the `L` node using a generic scheme. Any scalars required for evaluating the predicate in a `WHILE` loop or used otherwise in the body of the loop are assumed to be initially duplicated on all processors allocated to the loop. The arrays and scalars are then redistributed to the processors executing the tasks that actually need them. In a similar manner, at the end of the loop, we assume that all arrays are distributed and all scalars are duplicated across all processors executing the loop.

The routine used for updating cost information in a `C` node is shown in Figure 3.19. The essence of our technique is to use weights to capture the effects of the `THEN` and `ELSE` parts of a conditional statement. We first update costs for the MDG `THEN` part. We then use a technique similar to the one used for loops to obtain α_{if} and τ_{if} that model the processing cost for the `THEN` part. After that, we obtain a α_{else} and τ_{else} pair for the `ELSE` part. Now

UpdateCostsForUserDefNode($MDGNode$)
UpdateCostsForMDG($mdg[MDGNode]$)
for $i \leftarrow 0$ **to** $\log_2(NumProcs)$
do $X[i] \leftarrow 2^i$
    $Y[i] \leftarrow \mathrm{HSAS}(mdg[MDGNode], X[i])$
LinearRegression(X, Y, i, α, τ)
$alpha[MDGNode] \leftarrow \alpha$
$tau[MDGNode] \leftarrow \tau$

Figure 3.20 Filling in User-Defined Node Cost Parameters

we use an estimated execution frequency for the `THEN` part in order to compute α and τ for the `C` node as a whole. This estimated execution frequency is assumed to be obtained using profiling. With regards to data distribution, we again use a technique similar to the one for `L` nodes. We assume that all the data needed is gathered on all of the processors allocated to the `C` node initially and then distributed to the appropriate processors. Similarly, all data is again gathered on all of the processors at the end of the execution of the conditional. As in the case of loops, a generic data distribution scheme is used for arrays while scalars are duplicated.

Figure 3.20 shows the routine for computing the cost of a `U` node. The technique for determining the α and τ parameters and data distributions is similar to the one discussed for `L` and `C` nodes.

3.4 Summary

In this chapter, we considered the MDG representation for an application. Extensions to HPF and MATLAB to enable extraction of the MDG structure were discussed in Sections 3.2.1 and 3.2.2. In Section 3.2.3, we considered algorithms to extract the structure of the MDG from an extended HPF or MATLAB program. Models for node and edge costs in the MDG were presented in Sections 3.3.1 and 3.3.2. Finally, in Section 3.3.5, we discussed algorithms for computing the parameters for these cost models for all nodes and edges in the MDG.

The MDG captures information about available task and data parallelism in an application. This information is used by our allocation and scheduling algorithms discussed in the next chapter to decide on an execution scheme for the application.

CHAPTER 4

ALLOCATION AND SCHEDULING ALGORITHMS

4.1 Introduction

As previously discussed, our approach for the simultaneous exploitation of task and data parallelism relies on our allocation and scheduling algorithms. These algorithms are responsible for deciding on an execution scheme for an application given its Macro Dataflow Graph (MDG) representation (Chapter 3). The MDG provides detailed cost information for an application. As we will see in the rest of this section, these cost models are extensively used by our allocation and scheduling algorithms.

The allocation and scheduling algorithm used for simple MDGs (MDGs with only `S` nodes) forms the basis for the allocation and scheduling algorithm used for hierarchical MDGs. Therefore, we first discuss the allocation and scheduling algorithm for simple MDGs in Section 4.2. Later, in Section 4.3, we consider the allocation and scheduling algorithm for hierarchical MDGs.

4.2 Simple MDGs

For simple MDGs, we initially developed the Two Step Allocation and Scheduling (TSAS) algorithm described in Section 4.2.1. This algorithm views allocation and scheduling as independent problems and provides solutions for each. We have theoretically analyzed and quantified the performance of the TSAS algorithm. This analysis is provided in Section 4.2.2.

In many cases, however, the TSAS approach does not work very well. Consider the simple MDG shown in Figure 4.1(a). Figure 4.1(b) shows the allocation and resulting schedule produced by the TSAS approach. By considering allocation as an independent problem, the TSAS algorithm computes an allocation that does not account for the possible scheduling conflict between node 2 and nodes 3 or 4, thus, resulting in idle time in the schedule. If we account for the scheduling conflict by using constraints on allocation, we obtain an allocation that results

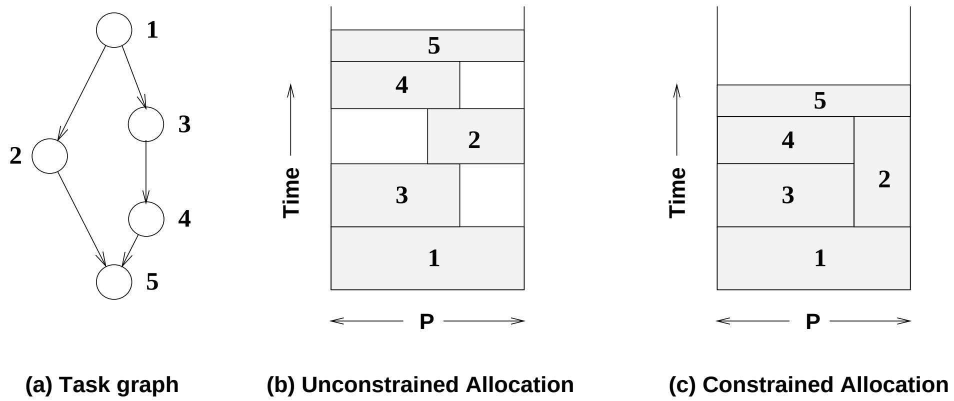


Figure 4.1 Example to Motivate Constraint Based Allocation

in a shorter schedule length. This is shown in Figure 4.1(c). Such constrained allocation is the idea behind our newer Simultaneous Allocation and Scheduling (SAS) algorithm described in Section 4.2.3.

4.2.1 Two Step Allocation and Scheduling (TSAS)

Given an MDG and a P processor system, our Two Step Allocation and Scheduling (TSAS) algorithm consists of the following steps

(1) Allocate processors to nodes in the MDG using the Convex Programming Allocation Algorithm.

(2) Schedule the allocated nodes using the Prioritized Scheduling Algorithm.

4.2.1.1 Convex Programming Allocation Algorithm (CPAA)

Given an MDG and a P processor system, let p_i be the number of processors allocated to the ith node. The CPAA computes the allocation for the MDG ($p_i \ \forall \ i = 1, n$) using the following steps

(1) Obtain a real number solution to the following minimization problem

$$\text{minimize } \Phi = \max(A, C) \tag{4.1}$$

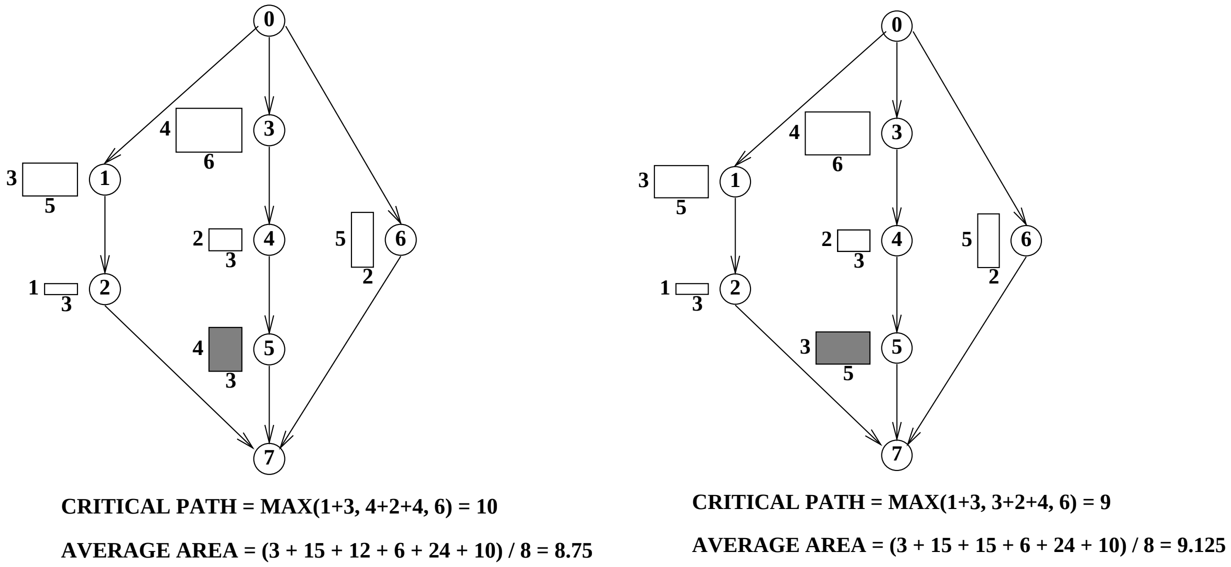
$$\textit{subject to } 1 \leq p_i \leq P \ \forall \ i = 1, n \tag{4.2}$$

where, the quantities A and C are the average area and the critical path for the MDG (defined in Section 3.3.4). Let the values of A and C that produce the minimum value of Φ be denoted by A_p and C_p, respectively.

(2) Set $p_i = \lfloor (p_i) \rfloor \; \forall \; i = 1, n$.

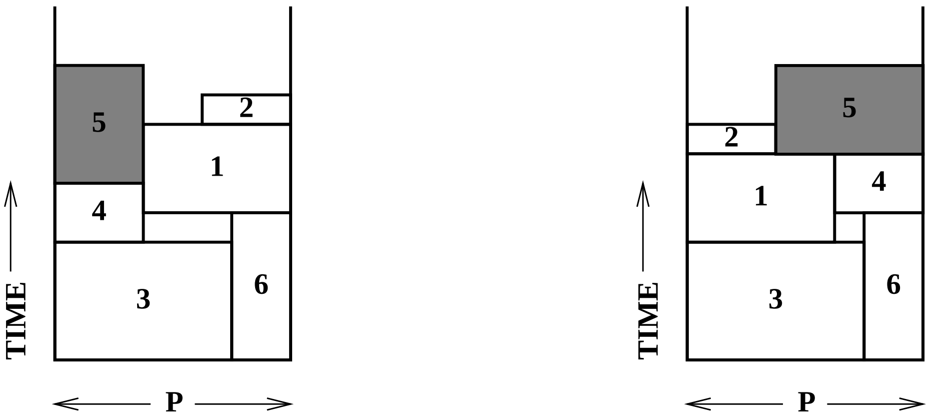
(3) Set $p_i = \min(p_i, PB)$, where PB is a function of P and can be computed using Corollary 4.1.

The intuition behind minimizing Φ is that it represents a theoretical lower bound on the time required to execute the application corresponding to a given MDG. The reason is that the execution time for the application can neither be smaller than the critical path of the MDG nor be less than the average area of the MDG. To better understand these constraints on the execution time, we can view a given processor system to be a bin of width P and each node in the MDG to correspond to a box of width p_i (processors allocated to node) and height T_i (cost of node as described in Section 3.3.3. The execution time of the application is the height of the processor system bin that accommodates boxes corresponding to all the nodes in the MDG while preserving all precedence constraints. Clearly, this height can neither be less than that of the tallest stack of boxes arising from the set of precedence constraints nor have area less than that required to accommodate all the boxes. The critical path of the MDG represents the former constraint and the average area of the MDG represents the latter constraint. Figure 4.2 illustrates the effect of the critical path and average area on the execution time of an example application. In this figure, we show two possible allocation schemes for the MDG corresponding to the application. We have shown a box corresponding to each node in the MDG; the width of the box is the allocation (p_i) for the node and the height of the box is the cost (T_i) of the node. Given the allocations and costs for nodes and ignoring edge costs, we have computed and shown the average area and critical path for an eight processor system using the two allocation schemes. As we can see, the first scheme has a higher critical path value and the second scheme has a higher average area value. This means the execution time for the first scheme is constrained by the critical path in the first case and by the average area in the second case. In terms of our processor system bin visualization, this translates to the height of the bin being constrained by the tallest stack of boxes in the first case and by the area required to accommodate all boxes in the second case. This is illustrated in Figure 4.3.



(a) Dominating Critical Path (a) Dominating Average Area

Figure 4.2 Effect of Allocation on Critical Path and Average Area



(a) Dominating Critical Path (a) Dominating Average Area

Figure 4.3 Processor System Bin Visualization of Program Execution

By obtaining an allocation (set of p_is) that concentrates on minimizing the lower bound Φ, we are hoping to minimize the execution time for the given application. However, our minimization process is carried out in a continuous domain, which results in the p_is being real numbers rather than integers, thus, necessitating rounding-off of the real number solution. We hope that this rounding-off produces an allocation close enough to the optimal allocation in the discrete domain. The reasons for using a continuous domain formulation are

(1) The posynomial properties of our cost models (Section 3.3) make our minimization problem equivalent to a convex programming formulation A. This guarantees finding the optimum minimum solution.

(2) A convex programming formulation has been shown to be solvable in polynomial time [68],[69] – this means we can find a optimum continuous allocation in polynomial time.

Note that in Step 1 of the CPAA, our search space for a set of p_is is not constrained in any form, tasks are allowed to use even the complete system if needed. This unconstrained minimization does not ensure "schedulability" - by this term we mean that allowing a task to use any number of processors may result in an inability to schedule tasks that can run concurrently with it. To improve the theoretical schedulability, we impose a bound PB on the allocation in Step 3. The optimum value for PB is computed using the analysis that follows in Section 4.2.2. As we shall see in Section 4.2.3, our Simultaneous Allocation and Scheduling (SAS) algorithm adopts a more practical approach to improving schedulability by modifying Step 1 of the CPAA.

4.2.1.2 Prioritized Scheduling Algorithm (PSA)

The steps involved in the PSA are

(1) Place the START node of the MDG in a queue called the Ready queue and mark its Earliest Start Time (EST_1) as 0.

(2) Pick a node i from the Ready queue that has the lowest value of the Earliest Start Time (EST_i). Use the schedule built so far to check the time at which the processor requirement of the task can be met. This is called the Processor Satisfaction Time (PST_i) for the node. Schedule the node at $\max(EST_i, PST_i)$ and compute its finish time (FT_i).

(3) If the node scheduled in the previous step is STOP, terminate the scheduler with a finish time FT_n; otherwise, proceed to the next step.

(4) Check all the successors of the node just scheduled to see if all the precedence constraints of any of them have been met, i.e., all their predecessors have been scheduled. Any successor nodes that meet this criterion are placed on the Ready queue with their Earliest Start Time computed.

(5) Repeat starting at Step 2.

Note that picking the node with lowest Earliest Start Time in Step 2 of the algorithm creates a priority among nodes, hence the name for the algorithm. The scheduler described above is a variant of the popular list scheduling algorithm that has been used by numerous researchers including Liu [70], Garey, Graham and Johnson [71], and Wang and Cheng [72]. Some of these researchers have also used variants of the basic list scheduling algorithm. When the maximum number of processors used by any node of the MDG is bounded, we can show that the PSA produces a schedule that finishes within a factor of the optimum (Theorem 4.1).

The complexity of the CPAA step has been shown to be $O(n^{2.5})$ [68],[69] and the complexity of the PSA is $O(np \log p)$. The $p \log p$ term arises in Step 2 of the PSA where the processor finish times have to be sorted to determine a suitable set of processors for the next task. Therefore, the complexity of the TSAS algorithm is $O(n^{2.5} + np \log p)$.

In the next section, we provide a theoretical analysis of the TSAS algorithm. We bound the solution produced by the TSAS in terms of Φ. As mentioned before, Φ represents a theoretical lower bound on the finish time obtainable for an MDG.

4.2.2 Theoretical analysis of TSAS

While developing the Convex Programming Allocation Algorithm (CPAA) discussed in the previous section, we used a continuous domain formulation and ignored the scheduling problem. As stated before, this formulation provides us not only with a starting point for obtaining a solution to the allocation and scheduling problem, but also provides us with a quantity Φ which represents a theoretical lower bound on the finish time that can be obtained for a given MDG. In this section, we provide some theoretical results that quantify the deviation of the solution obtained using our Two Step Allocation and Scheduling (TSAS) algorithm with respect to Φ. The bound on this deviation is derived in three phases:

- Theorem 4.1 examines the possible deviation of the solution produced by our Prioritized Scheduling Algorithm (PSA) with respect to the solution produced by an optimal scheduling algorithm. This deviation is dependent on the maximum number of processors (PB) used by any node in the MDG; the lower the value of PB, the smaller the deviation. The intuitive reason for this is that the maximum idle time possible in a schedule built by the PSA is dependent on the value of PB. This will become clearer in the discussion of the proof of Theorem 4.1.

- Theorem 4.2 examines the effect of rounding-off (Step 2 of CPAA) and bounding (Step 3 of CPAA) on the value of Φ. Rounding-off and bounding modify the values of the p_is computed in Step 1 of the CPAA; this may result in a change in the values of the critical path and average area for the MDG and thus, a change in the value of Φ. The lower the value of the bound PB, the larger the change in Φ. Again, the discussion of the proof of Theorem 4.2 will make this clearer.

- Theorem 4.3 summarizes the effects discussed in Theorems 4.1 and 4.2 to quantify the possible deviation of the allocation and scheduling solution produced by the TSAS with respect to Φ, which represents a theoretical lower bound on the finish time. This deviation can be seen to be dependent on PB; Corollary 4.1 selects a value for PB that minimizes it.

In constructing proofs for the theorems below, we have assumed that the underlying computation and communication cost functions are of the form discussed in Section 3.3.

We present below a definition of a term used in the proof of the theorem that follows.

Definition 4.1 *Area of Useful Work*

When a schedule S is used for a given MDG on a given multicomputer system, the area of useful work (W_s) done by it is defined as

$$W_s = \sum_{i=1,n_b} t^i_{busy} \cdot p^i \tag{4.3}$$

where t^i_{busy} is the ith interval during which a constant number (p^i) of processors are kept busy by the schedule. The quantity n_b denotes the total number of such intervals.

Theorem 4.1 *Assume we are given an MDG with n nodes and a processor allocation such that no node uses more than PB processors. Let T_{psa} denote the value of the finish time obtained by scheduling this MDG on a given p processor system using the PSA algorithm and T^{PB}_{opt} denote the value obtained using the best possible scheduler. The relationship between these two quantities is given by*

$$T_{psa} \leq (1 + \frac{P}{p - PB + 1}) \cdot T^{PB}_{opt} \tag{4.4}$$

Proof:

In the best case, the area of useful work done by the optimal scheduling algorithm can be $p \cdot T^{PB}_{opt}$, because the optimal algorithm can, at best, keep all p processors in the system busy for the entire length of the schedule it produces. If the work done by the PSA is denoted by W_{psa}, we can write

$$W_{psa} \leq p \cdot T^{PB}_{opt} \tag{4.5}$$

If any node uses at most PB processors, we can say that if the PSA is unable to schedule the next node immediately, it has at least $p - PB + 1$ processors currently busy. However, as we will see later, this will not always be true. If the duration when this is not true is Δ (in the worst case), we can write (using the definition of useful work)

$$W_{psa} \geq (T_{psa} - \Delta) \cdot (p - PB + 1) + W_{\Delta} \tag{4.6}$$

Here we are assuming W_{Δ} is the worst case useful work (if any) done during the periods when less than $p - PB + 1$ processors are busy.

If more than PB processors are idle, it means the PSA algorithm has a case when $PST < EST$ for all the unscheduled nodes (Step 2 of PSA). This implies that every other unexecuted node is dependent on ongoing events which may include a node execution or an edge delay. It is also clear that such a situation could occur many times in the building of a schedule.

Let us call a situation such as the one described above an Idling Situation (IS). We now contend that one or more of the events involved in the ith such IS (IS_i) control each of the the events of every subsequent IS (IS_j *for all* $j > i$). If this were not true, it means that we can find some node execution or edge delay in an $IS_k, k > i$ such that no event in IS_i controls it. In such a case this node execution or edge delay would have been scheduled concurrently with the events in IS_i, which means it cannot belong to IS_k, which is a contradiction. Therefore our contention is true.

The implication of this dependence between events in ISs is that they must form a set of paths (partial or complete) in the given MDG. We know that the length of any path in the MDG is bounded by the length of the critical path. Therefore, in the worst case, we can see that the total duration for which IS's can occur in the schedule is the length of the critical path. Since T_{opt}^{PB} must be at least the length of the critical path, we can write

$$\Delta \leq T_{opt}^{PB} \tag{4.7}$$

In the worst case, no processors will be busy during any IS (all events are edge delays), implying no work is done. This would give us a $W_{\Delta} \geq 0$. Using this inequality and Equation (4.7) in Equation (4.6), we obtain

$$W_{psa} \geq (T_{psa} - T_{opt}^{PB}) \cdot (p - PB + 1) \tag{4.8}$$

From Equations (4.5) and (4.8), we have

$$(T_{psa} - T_{opt}^{PB}) \cdot (p - PB + 1) \leq W_{psa} \leq T_{opt}^{PB} \cdot p \tag{4.9}$$

$$\Rightarrow T_{psa} \leq (1 + \frac{P}{P - PB + 1}) \cdot T_{opt}^{PB} \tag{4.10}$$

which is the required result $\Box$.

Theorem 4.2 *In Steps 2 and 3 of the CPAA we modify the processor allocation produced by Step 1 of the CPAA. If* T_{opt}^{PB} *denotes the value of the finish time obtained for the given MDG on a* p *processor system with this modified allocation using an optimal scheduling algorithm, the relationship between* T_{opt}^{PB} *and* Φ *is given by*

$$T_{opt}^{PB} \leq (\frac{2 \cdot P}{PB}) \cdot \Phi \tag{4.11}$$

where Φ is the solution obtained in Step 1 of the CPAA and represents a theoretical lower bound on the finish time obtainable for the given MDG.

Proof: By closely examining Steps 2 and 3 of the CPAA, we see that the modification of the allocation produced in Step 1 can only result in a decrease in allocation for any node in the MDG. Let us now examine the effect of a decrease in allocation for a node on the values of the average area and the critical path. The effect can be seen from the definition of these quantities in Section 3.3.4 and the cost functions of Sections 3.3.1 and 3.3.2.

From this information, we can see that decreasing the processor allocation for any node will only decrease the value of the average area. Again, by looking closely at the cost functions, we see that decreasing the processor allocation of a node i from p_i to p_i' increases the critical path. $\frac{p_i}{p_i'}$. This factor arises due to the structure of t_i^C; we are also assuming that data redistribution cost components are negligible when compared with the processing cost.

Having seen this, we now examine the effect of Steps 2 and 3 of the CPAA on the values of the average area and critical path produced by Step 1 (A_P and C_P).

Since we round-off each p_i using a *floor* function in Step 2, it can be shown that the processor allocation for the ith node is changed by twice its original value in the worst case. This happens when p_i is changed from 1.99 to 1 (the average change will not be as drastic). Let the value of the average area and critical path time of the MDG thus allocated be denoted by A_{RO} and C_{RO}, respectively. From the discussion on the effect of decrease in processor allocation, we can write:

$$A_{RO} \leq A_P \tag{4.12}$$

$$C_{RO} \leq 2 \cdot C_P \tag{4.13}$$

In the bounding step (Step 3), it can be shown that the worst-case decrease in processor allocation for the i node is from P to PB. If A_{PB} is the value of the average area and C_{PB} is the value of the critical path time for the bounded allocation, using the discussion on effects of decrease in processor allocation, we have

$$A_{PB} \leq A_{RO} \tag{4.14}$$

$$C_{PB} \leq \frac{P}{PB} \cdot C_{RO} \tag{4.15}$$

Since T_{opt}^{PB} denotes the time obtained by using the optimal scheduler on this rounded-off and bounded processor allocation, in the best possible situation, we have

$$T_{opt}^{PB} = \max(A_{PB}, C_{PB}) \tag{4.16}$$

Using Equations (4.12), (4.14) and (4.16), we have

$$T_{opt}^{PB} \leq \max(A_P, (\frac{2 \cdot P}{PB}) \cdot C_P) \tag{4.17}$$

From the equation above and the definition of Φ in Section 4.2.1, we have

$$T_{opt}^{PB} \leq (\frac{2 \cdot P}{PB}) \cdot \Phi \tag{4.18}$$

which is the required result $\square$.

Theorem 4.3 *Let T_{TSAS} denote the value of the finish time obtained for allocation and scheduling using the* TSAS *algorithm (Section 4.2.1). The deviation of T_{TSAS} from Φ is given by*

$$T_{TSAS} \leq (1 + \frac{P}{P - PB + 1}) \cdot (\frac{2 \cdot P}{PB})\Phi \tag{4.19}$$

where Φ represents a theoretical lower bound on the finish time obtainable for the given MDG and is computed in Step 1 of the CPAA.

Proof: T_{TSAS} is the finish time produced at the end of the scheduling step (Step 2) of the TSAS algorithm. This quantity is the same as the finish time produced by the PSA algorithm. Theorem 4.1 describes the relationship between the finish time produced by the PSA algorithm and the finish time obtainable using an optimal scheduling algorithm given an allocation. Theorem 4.2 in turn describes the relationship between the finish time obtainable using an optimal scheduling algorithm and Φ after the allocation step (Step 1) of the TSAS. Tying in the results of Theorems 4.1 and 4.2, we obtain the required result $\Box$.

Corollary 4.1 *The optimum value of PB to use for the* TSAS *algorithm given P processors is the solution to the following minimization problem:*

$$minimize\ (1 + \frac{P}{P - PB + 1}) \cdot (\frac{2 \cdot P}{PB}) subject\ to\ 1 \leq PB \leq P \tag{4.20}$$

Proof: This corollary follows from Theorem 4.3 $\Box$.

4.2.3 Simultaneous Allocation and Scheduling (SAS)

As mentioned in Section 4.2.1, Step 1 of the Convex Programming Allocation Algorithm (CPAA) does not model the scheduling process. In that section we suggested a theoretical technique to ensure schedulability of the allocation produced. In this section we discuss a new Simultaneous Allocation and Scheduling (SAS) algorithm that uses constrained allocation to improve schedulability. The SAS consists of the following steps

(1) Allocate P processors to all the nodes in the MDG and schedule using the Modified Prioritized Scheduling Algorithm.

(2) Note the finish time of the schedule produced.

(3) Allocate processors to nodes in the MDG using the Constrained Convex Programming Allocation Algorithm.

(4) Schedule the allocated nodes using the Modified Prioritized Scheduling Algorithm.

(5) Check the finish time of the schedule to see if it improves the best finish time; if it does, store the allocation found and update the best finish time.

(6) Check if the current solution has occurred before; if it has not, repeat steps starting at Step 3. If solution has been seen before, terminate.

4.2.3.1 Constrained Convex Programming Allocation Algorithm (CCPAA)

The CCPAA for a given MDG and a given P processor system has the following steps

(1) Obtain a real number solution to the following minimization problem

$$\text{minimize } \Phi = \max(A, C)$$

$$\textit{subject to } 1 \leq p_i \leq P \ \forall\ i = 1, n \tag{4.21}$$

$$\textit{and constraint set } CS \ : \sum_{i \in CURR_j} p_i \leq P \ \forall\ j = 1, NCON$$

The description of the MPSA algorithm that follows will clarify the concept of the constraint set.

(2) Set $p_i = \lfloor (p_i) \rfloor \ \forall\ i = 1, n$.

4.2.3.2 Modified Prioritized Scheduling Algorithm (MPSA)

The steps involved in the MPSA are

(1) Place the START node of the MDG in a queue called the *Ready* queue and mark its *Earliest Start Time* (EST_1) as 0. Initialize the number of constraints ($NCON$) to 0 and the constraint set (CS) to $\{\emptyset\}$.

(2) Pick a node i from the ready queue that has the lowest value of the earliest start time (EST_i). Use the schedule built so far to check the time at which the processor requirement of the task can be met. This is called the *Processor Satisfaction Time* (PST_i) for the node. Schedule the node at $\max(EST_i, PST_i)$ and compute its finish time (FT_i).

(3) If, in the previous step, the node i is scheduled at PST_i, i.e. $PST_i > EST_i$, increment $NCON$ and add $CURR_{NCON}$ to the constraint set CS, where $CURR_{NCON}$ is given by

$$CURR_{NCON} = \{j \mid FT_j \geq PST_i\} \tag{4.22}$$

Note that node i is included in $CURR_{NCON}$ and that nodes considered for inclusion are only those that have already been scheduled. [1]

[1] The $CURR$ set for our example of Figure 4.1(b) would be $2, 3, 4$.

(4) If the node scheduled in the previous step is STOP, terminate the scheduler with a finish time FT_n; otherwise, proceed to the next step.

(5) Check all the successors of the node just scheduled to see if all the precedence constraints of any of them have been met, i.e., all their predecessors have been scheduled. Any successor nodes that meet this criterion are placed on the Ready queue with their Earliest Start Time computed.

(6) Repeat starting at Step 2.

4.2.3.3 Comments

The basis for the SAS algorithm is to model scheduling conflicts in the allocation process, thereby making the allocation more "schedulable." As the scheduling conflicts depend on the allocation, we iterate through the allocation and scheduling steps a few times to obtain a good solution. Instead of iterating in this manner, we could terminate when the finish time begins to increase. However, during our experiments we found that a temporary increase in the schedule time in one iteration can sometimes lead to a big decrease in the next iteration. In practice, solutions repeat after a few iterations; in our experiments, we have observed that the repetition of solutions happens in under ten iterations.

The constraints we apply to our minimization problem can be easily shown to be posynomial functions. This leads to our minimization still being equivalent to a convex programming formulation, thus providing us the benefits outlined in Section 4.2.1. As the results provided in the next section show, the SAS algorithm appreciably improves the finish times obtained for benchmark MDGs when compared to the TSAS algorithm.

The addition of constraints does not change the complexity of solving a convex programming formulation; therefore, the complexity of the CCPAA step remains $O(n^{2.5})$ [68][69]. Again, the the complexity of the MPSA remains $O(np \log p)$ since Step 2 is still the dominating step in the algorithm. Therefore, the complexity of the SAS algorithm is $O(X(n^{2.5} + np \log p))$. As we have discussed before, a value of 10 for X is suitable for good quality results, which makes the complexity of the SAS algorithm $O(n^{2.5} + np \log p)$ for all practical purposes.

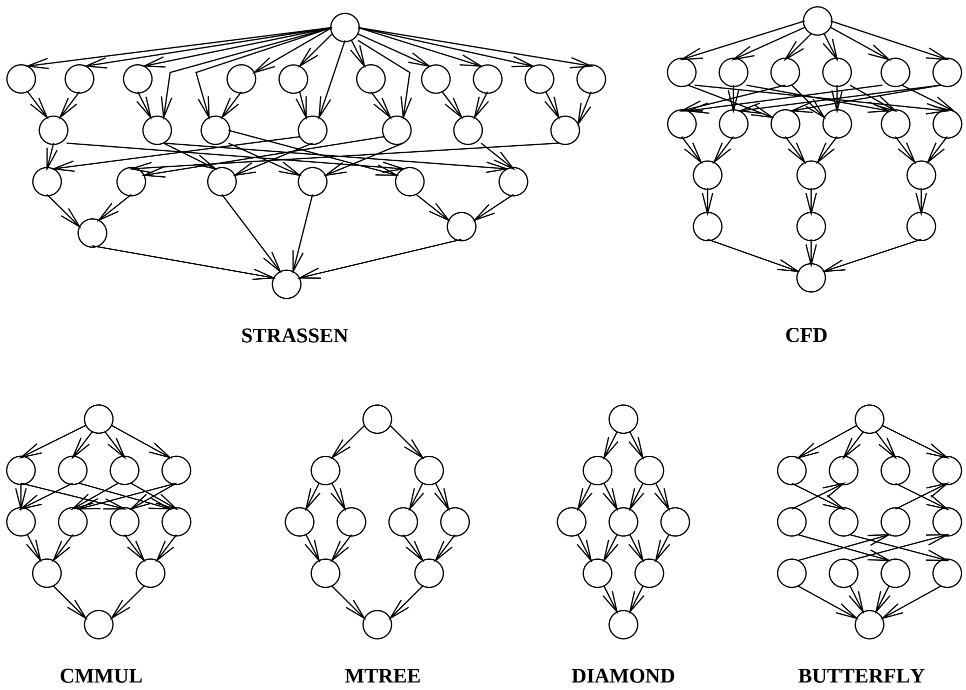


Figure 4.4 Some Benchmark MDGs.

4.2.4 Results

To compare the performance of the TSAS and SAS algorithms, we measured the predicted speedups obtained for both algorithms using a set of eight benchmark MDGs. Three of the benchmarks – Cmmul, Strassen and Cfd – correspond to actual programs while the other five – Butterfly, Diamond, Mtree, Random1 and Random2 – are synthetic benchmarks. The reason for using synthetic MDGs was to ensure the robustness of the algorithms. The structure of the MDGs corresponding to a few of the benchmarks is shown in Figure 4.4.

We used three system sizes for our experiments - 32, 64 and 128 processors. The results of our experiments are shown in Table 4.1. In addition to tabulating the speedups for the TSAS and SAS algorithms, we have also shown the improvement obtained by using the SAS algorithm in the last column. From this column we can see that the SAS algorithm outperforms the TSAS algorithm in a number of cases. This makes a strong case for using the SAS algorithm.

Table 4.1 Comparison of Speedups Obtained Using the TSAS and SAS Algorithms

Benchmark	System Size	Speedup		Improvement
		S_{TSAS}	S_{SAS}	$(\frac{S_{SAS}}{S_{TSAS}})$
CMMUL	32	19.0	19.0	1.00
	64	28.7	29.8	1.04
	128	41.6	42.7	1.03
STRASSEN	32	20.3	20.3	1.00
	64	33.9	33.9	1.00
	128	52.9	57.0	1.08
CFD	32	14.9	23.7	1.59
	64	28.4	42.3	1.49
	128	46.1	68.0	1.48
BUTTERFLY	32	14.3	21.3	1.49
	64	22.8	35.0	1.54
	128	35.8	50.1	1.40
DIAMOND	32	12.0	15.1	1.26
	64	15.1	19.0	1.25
	128	17.5	21.6	1.24
MTREE	32	14.6	19.3	1.32
	64	26.7	33.3	1.25
	128	36.9	51.2	1.39
RANDOM1	32	15.2	17.4	1.14
	64	21.3	26.6	1.25
	128	26.4	31.8	1.21
RANDOM2	32	15.7	17.7	1.13
	64	21.1	24.1	1.14
	128	25.1	28.8	1.15

```
HSAS(MDG, NumProcs)
FinishTime ← SAS(MDG, NumProcs)MDGNodeList ← nodes[MDG]
while MDGNodeList ≠ NIL
        do MDGNode ← node[MDGNodeList]
            switch
                case type[MDGNode] = LoopNode :
                    if looptype[MDGNode] = While
                        then HSAS(loopmdg[MDGNode], allocation[MDGNode])
                        else  HSASForLoop(loopmdg[MDGNode], allocation[MDGNode])
                case type[MDGNode] = ConditionalNode :
                    HSAS(ifmdg[MDGNode], allocation[MDGNode])
                    HSAS(elsemdg[MDGNode], allocation[MDGNode])
                case type[MDGNode] = UserDefNode :
                    HSAS(mdg[MDGNode], allocation[MDGNode])MDGNodeList ← next[MDGNodeL

return FinishTime
```

Figure 4.5 Allocation and Scheduling for Hierarchical MDGs

4.3 Hierarchical MDGs

The Simultaneous Allocation and Scheduling (SAS) algorithm described in Section 4.2.3 forms the basis of our hierarchical allocation and scheduling algorithm. Basically, we use the SAS algorithm to allocate and schedule at each level in a hierarchical MDG. Figure 4.5 shows the Hierarchical Simultaneous Allocation and Scheduling (HSAS) algorithm. The HSAS algorithm first uses the SAS algorithm to allocate and schedule at the current level. Then, it checks for the presence of `L`, `C` and `U` nodes at the current level. Depending on the type of node, the following actions are taken:

`L` nodes If the loop is a `WHILE` loop, we simply use the HSAS algorithm to allocate and schedule the MDG corresponding to the loop body. However, if the loop is a `FOR` loop, we use the algorithm shown in Figure 4.6 to allocate and schedule the loop body. This algorithm uses unrolling to try to increase available task parallelism to improve the performance. Figure 4.7 shows an example of a loop, the MDG corresponding to the body of the loop and the effect of unrolling the loop once. As we can see, the MDG corresponding to the unrolled loop has more task parallelism; this can lead to potential performance gains. However, unrolling can also lead to code expansion, which is why we use the parameters `MAXFACTOR` and `IMPFACTOR` in Figure 4.6 to limit the amount of unrolling. Currently, we

```
HSASForLoop(MDG, NumProcs)
FinishTime ← SAS(MDG)
OldFinishTime ← ∞
UnrollFactor ← 0
while (UnrollFactor < MaxFactor) and (FinishTime ≤ ImpFactor × OldFinishTime)
        do OldFinishTime ← FinishTime
           UnrollFactor ← UnrollFactor + 1
           TempMDG ← UnrollMDG(MDG, UnrollFactor)
           FinishTime ← SAS(MDG)/UnrollFactor
Parent ← parent[MDG]
loopmdg[Parent] ← TempMDG
unrollfactor[Parent] ← UnrollFactor − 1
return OldFinishTime
```

Figure 4.6 Allocation and Scheduling for Loop (`L`) Nodes

use a value of 4 for `MAXFACTOR` and 0.95 for `IMPFACTOR`, which means we limit unrolling to a maximum of four times and increase the amount of unrolling only if the finish time is reduced to 95% of its original value.

`C` **nodes** For conditional nodes, allocation and scheduling is done using the HSAS algorithm on the MDGs corresponding to the bodies of the `IF` and `ELSE` parts.

`U` **nodes** In the case of user-defined nodes, we use the HSAS algorithm for allocating and scheduling the MDG corresponding to the body of the user-defined function.

As can be seen, allocation and scheduling at lower levels is done using the allocation already computed at a preceding higher level.

4.4 Summary

In this chapter we considered the problem of allocation and scheduling given the MDG representation for an application. Allocation and scheduling exploit available task and data parallelism in an application and decide on a suitable execution scheme. We first considered allocation and scheduling for simple MDGs in Section 4.2 and later used these algorithms to construct allocation and scheduling algorithms for hierarchical MDGs in Section 4.3.

```
for i = 1:100
  read(ar);
  read(ai);
  fft(ar,ai);
  br=ar';
  bi=ai';
  fft(br,bi);
end
```

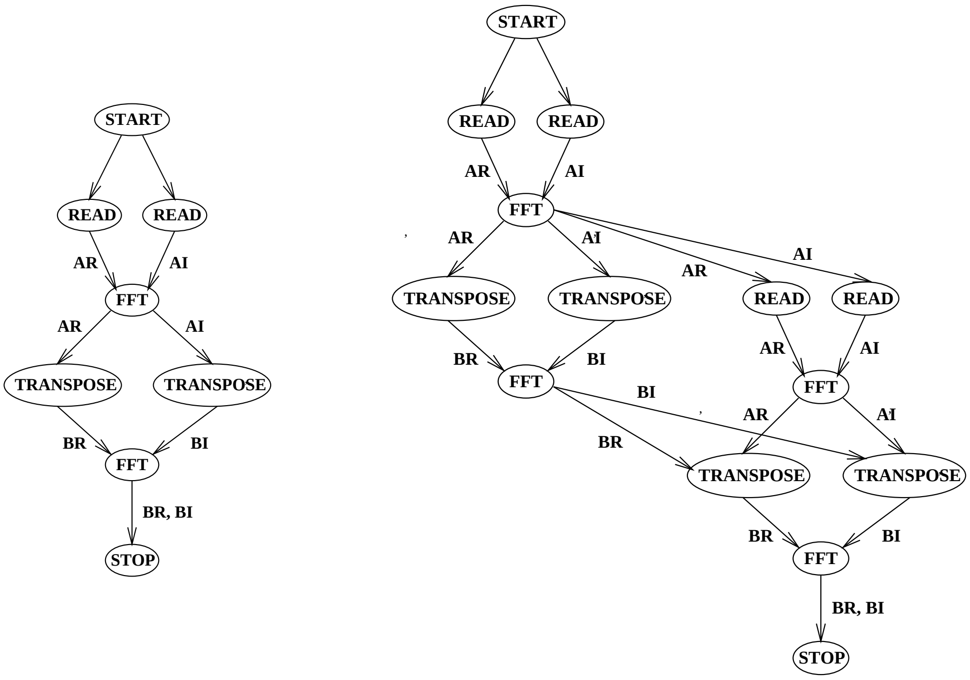


Figure 4.7 Example Illustrating Unrolling for Loops

Once we have decided on a suitable execution scheme for an application, in order to implement it, we have to generate MPMD code and provide run-time support for data redistribution. The problem of efficient data redistribution is the subject of Chapter 5.

CHAPTER 5

DATA REDISTRIBUTION

5.1 Introduction

As mentioned before, data redistribution is essential for implementing an execution scheme that exploits task and data parallelism. Individual tasks execute in a data parallel fashion on subsets of processors and data dependences between tasks may necessitate not only changing the set of processors an array is distributed on but also the scheme of distribution itself. Figure 5.1 illustrates the need for redistribution between a pair of tasks. `TaskA` is executed using five processors and writes to array `A`; `TaskB` is executed using three processors and reads from array `A`. This necessitates the redistribution of `A` from the five processors executing `TaskA` to the three processors executing `TaskB`. In addition to changing the set of processors `A` is distributed on, we could also potentially change the distribution scheme of `A` going from `TaskA` to `TaskB`. For instance, `TaskA` might use a `BLOCK` distribution for `A`, whereas `TaskB` might use a `CYCLIC` distribution.

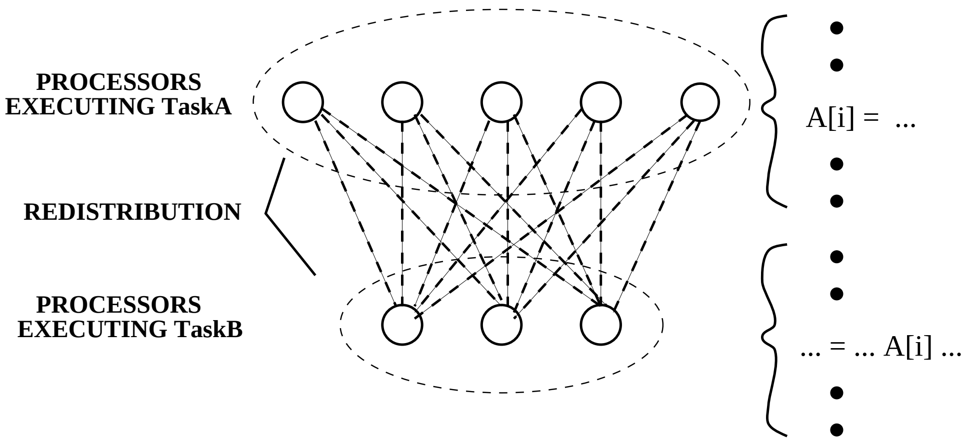


Figure 5.1 Example Illustrating the Necessity of Data Redistribution

Formally, we can define a data redistribution R to be the set of routines that – given an n-dimensional array A on a set of source processors P_s with source distribution D_s – transfer all the elements of the array to a set of target processors P_t with a target distribution D_t. In general, D_s and D_t can specify arbitrary data distributions along each dimension of the array. Therefore, a redistribution routine has to figure out exactly what data is to be sent/received by each source/target processor. It is possible to use a simple run-time resolution approach for redistribution. In this approach, each source processor does the following:

- Computes the indices of the elements it owns based on the source distribution.
- Uses the computed indices to compute the target processor for each index based on the target distribution.
- Packs the elements appropriately into buffers for each of the target processors.
- Sends the contents of its buffers to the target processors.

The target processors essentially do the reverse. However, such a run-time approach is very costly compared to a method like ours which makes use of the distribution information in a more intelligent manner.

The important features of our redistribution routines are

- The source and target processor sets can be any arbitrary subset of the given processor system. For example, we could redistribute from processors $\{0, 3, 4, 6\}$ to processors $\{1, 2\}$. Our implementation of the run-time system for the Paradigm compiler is based on the Message Passing Interface (MPI) and makes use of the concepts of process groups and communicators [73],[74] to handle arbitrary processor sets. For our MATLAB compiler, we base our implementation on the BLACS [40] library. This library provides message passing support for process groups.
- Arrays being redistributed can have any of the possible regular distributions on the source and target processor sets. Note that one of the most general cases of redistribution is `BLOCKCYCLIC(m)` to `BLOCKCYCLIC(n)`, where m and n are relatively prime. We handle all the types of distributions in a uniform manner.

- Arrays being redistributed can have an arbitrary number of dimensions. The complexity of our algorithms scales linearly with an increase in the number of dimensions.

- We handle multiple arrays being redistributed at the same time using message aggregation [24],[75]. This means that we have just one *Send-Recv* operation between a pair of processors even if there are elements from more than one array being communicated between them.

- Our redistribution algorithm generates local addresses for distributed arrays. In addition, it tries to maximize spatial memory locality during packing and unpacking.

- We use a simple communication scheduling scheme to reduce contention during redistribution.

Our redistribution techniques rely on a mathematical representation for regular distributions which we call FALLS (FAmiLy of Line Segments). Although many representations exist for regular data distributions on distributed memory machines [25],[76], we felt the need for a new representation for our work because none of the previous representations satisfied our requirements. A primary requirement for us was to be able to mathematically represent a regular distribution on any given subset of processors of the given system and not on the entire system as all the current representations do. In addition, we had to be able to perform redistribution communication analysis efficiently. This led to our development of the FALLS representation and algorithms for redistribution communication analysis based on it.

Section 5.2 discusses the FALLS representation along with our redistribution algorithm and some important optimizations. To demonstrate the efficiency and scalability of our algorithm, we present some experimental results in Section 5.3.

5.2 The FALLS Representation and Redistribution

To perform a redistribution, for a given source-target processor pair, one has to look at the set of elements owned by the source processor before redistribution (based on the source distribution) and the set of elements owned by the target processor after redistribution (based on the target distribution). The intersection of these sets determines the elements that have to be transferred between the pair of source-target processors. The FALLS representation is particularly useful in this context as it can be used to easily determine the two things most important for redistribution – which pairs of source-target processors need to communicate and the intersection of the data sets of such a pair of source-target processors.

For simplicity, we first develop the FALLS representation for regular distributions of a one-dimensional array in a step-by-step manner. We will later extend our ideas to multiple dimensions.

5.2.1 Line Segments (LS)

Consider a one-dimensional array A of size n. Fundamental to the FALLS representation is the idea of using Line Segments (LS) to represent a contiguous block of elements. An LS L can be represented by a pair of numbers (l, r). For our representation, this LS (in the context of array A) is taken to mean the block of elements of A with indices starting at l and ending at r (numbering $r - l + 1$). For example, the LS $L = (4, 5)$ has $5 - 4 + 1 = 2$ elements and represents the elements with indices 4 and 5. The quantity l is called the LOW of a line segment L; i.e., $l = LOW(L)$. Similarly, r is called the $HIGH$ of L ($r = HIGH(L)$). Note that a single element with index l has the LS representation (l, l).

Since our primary interest is in being able to find intersections for sets of elements, we see that the intersection of two LS's $L_1 = (l_1, r_1)$ and $L_2 = (l_2, r_2)$ (denoted by $LI_{L_1 \bigcap L_2}$) is given by

$$LI_{L_1 \bigcap L_2} = \begin{cases} (\max(l_1, l_2), \min(r_1, r_2)) & if \;\; \max(l_1, l_2) \leq \min(r_1, r_2) \\ \emptyset & otherwise \end{cases} \tag{5.1}$$

5.2.2 FAmiLy of Line Segments (FALLS)

The line segment notion can be extended to what we call a FAmiLy of Line Segments (FALLS). A FALLS F can be represented by a tuple (l, r, str, num). Intuitively, F represents a set of num equally spaced, equally sized blocks of elements; the first block starts at l and ends at r; the stride between successive l's is str. Note that these are nonoverlapping blocks. The ith ($0 \leq i \leq num - 1$) LS of F (denoted by L^i and called the ith member) is given by

$$L^i = (l + i \cdot str, r + i \cdot str)$$

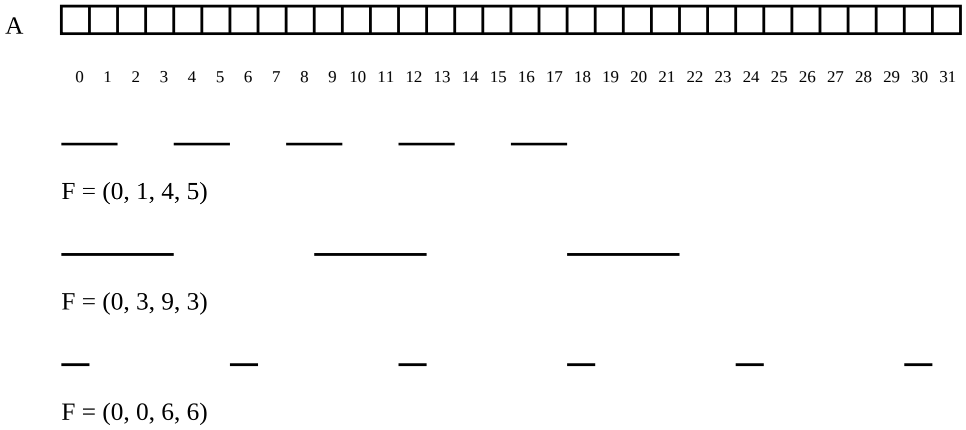


Figure 5.2 Examples of FALLS

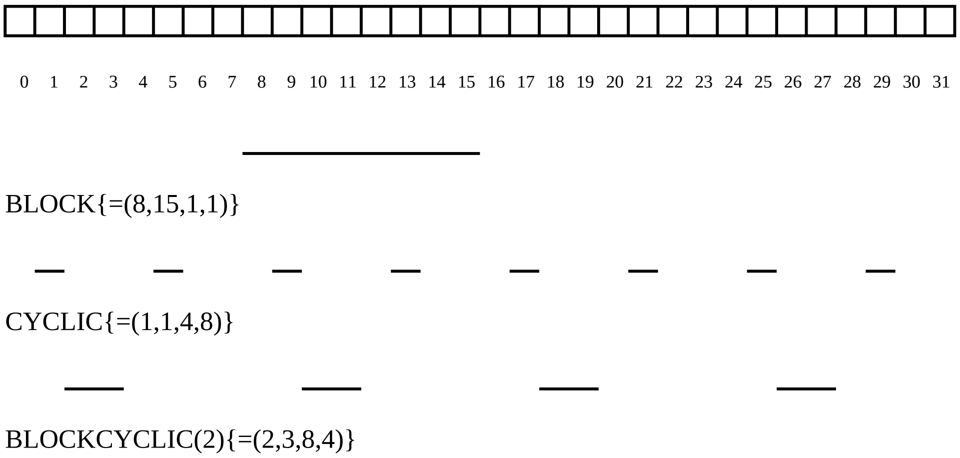


Figure 5.3 Examples of FALLS Describing Regular Distributions

Figure 5.2 shows a few examples of FALLS.

Using the notion of FALLS, it is possible to represent the set of elements of A owned by a particular processor under any regular distribution. Figure 5.3 shows the FALLS representation for elements owned by $P1$ in a four-processor system $(P0 - P3)$ for various distributions of A when $n = 32$.

Again, in the context of data redistribution, computing the intersection of two FALLS is of interest to us. We can represent the set of elements owned by the source and destination processors by a pair of FALLS; the intersection of the pair of FALLS determines the set of elements that have to be transferred between the source and destination processors.

BRUTEFORCE()

$FI_{F_1 \bigcap F_2} = \emptyset$
for $i_1 \leftarrow 0$ **to** $num_1 - 1$
do $L_1 = (l_1 + i_1 \cdot str_1, r_1 + i_1 \cdot str_1)$
    for $i_2 \leftarrow 0$ **to** $num_2 - 1$
    do $L_2 = (l_2 + i_2 \cdot str_2, r_2 + i_2 \cdot str_2)$
        $FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup LI_{L_1 \bigcap L_2}$

Figure 5.4 Brute Force FALLS Intersection Algorithm

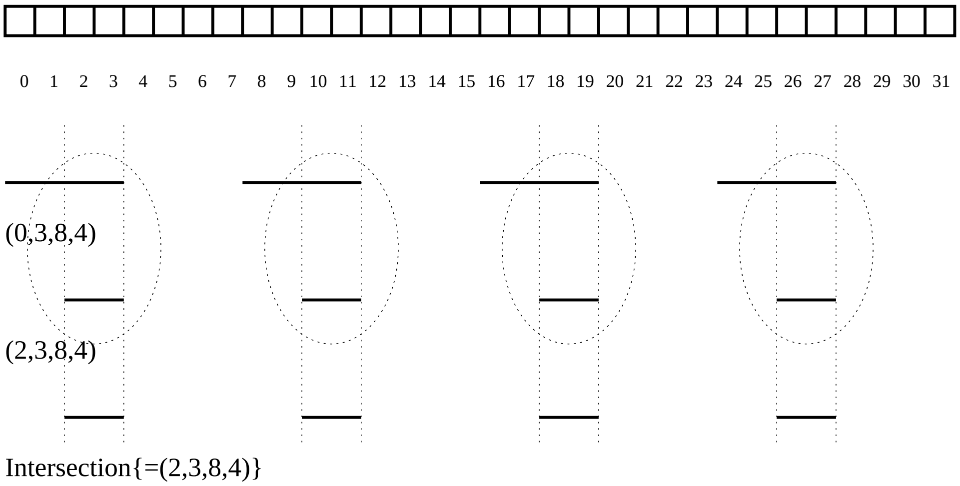


Figure 5.5 Example of FALLS Intersection

Given two FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$, a simple brute force algorithm to compute the intersection ($FI_{F_1 \bigcap F_2}$) is shown in Figure 5.4. This brute force approach considers every possible pair of members from the two FALLS and applies the LS intersection algorithm to them. This technique can be very inefficient by considering the example of FALLS intersection shown in Figure 5.5. In this example, there are just four non-empty intersections whereas our brute force algorithm would perform sixteen iterations.

There are a couple of important observations to make in our example of Figure 5.5. One of them is that the intersecting pairs of members of the two FALLS (circled in the figure) have the same relationship between them; i.e., their relative displacement is the same. The relative displacement R for a pair of members (i_1, i_2) of two FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$ is the distance between their LOWs and is given by

$$R = (l_1 + i_1 \cdot str_1) - (l_2 + i_2 \cdot str_2)$$

The notion of relative displacement also gives rise to the idea of periodicity in the relationships between members of the two FALLS. We can define a quantity called the intersection period to be the distance that separates pairs of members of two FALLS that have the same relative displacement. The length of the intersection period ($FP_{F_1 \bigcap F_2}$) for a given pair of FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$ can be written as

$$FP_{F_1 \bigcap F_2} = lcm(str_1, str_2)$$

We also find it convenient to define a pair of quantities called m_1 and m_2 as follows:

$$\begin{aligned} m_1 &= \frac{FP_{F_1 \bigcap F_2}}{str_1} \\ m_2 &= \frac{FP_{F_1 \bigcap F_2}}{str_2} \end{aligned} \tag{5.2}$$

Intuitively, these quantities represent the number of members of each FALLS occurring in a period. It can easily be verified that a pair of members from the two FALLS (i_1, i_2) will have the same relative displacement as the pair of members $(i_1 + m_1, i_2 + m_2)$. For the example of Figure 5.7, $FP_{F_1 \bigcap F_2} = 16$, $m_1 = 1$, and $m_2 = 4$.

Another observation to make in Figure 5.5 is that the intersection of the two FALLS in this case turns out to be a FALLS (as noted in the figure). These observations imply that we need only look at possible intersections between pairs of members of the two FALLS that occur within a period and extend any resulting intersections to all other periods (thus giving rise to a FALLS structure). This gives us a more efficient intersection computation algorithm shown in Figure 5.6. For the algorithm, (I_1, I_2) is the first pair of members of the two FALLS that intersect; all other terms have the same meaning as explained above. If we use this algorithm for the example of Figure 5.5, we see that $(I_1 = 0, I_2 = 0)$ is the first pair of members of the two families that intersect. As seen earlier, $FP_{F_1 \bigcap F_2} = 8$ and $m_1 = m_2 = 1$. This means our algorithm will iterate just once with $L_1 = (0, 3)$ and $L_2 = (2, 3)$ which gives us a non-empty intersection $LI_{L_1 \bigcap L_2} = (2, 3)$ making $l = 2$ and $r = 3$. Using the value 8 for $FP_{F_1 \bigcap F_2}$ gives

PERIODS()

$I_1 = \max(0, \lceil \frac{l_2 - r_1}{str_1} \rceil)$
$I_2 = \max(0, \lceil \frac{l_1 - r_2}{str_2} \rceil)$
$FI_{F_1 \bigcap F_2} = \emptyset$
for $i_1 \leftarrow I_1$ **to** $I_1 + m_1 - 1$
do
  $L_1 = (l_1 + i_1 \cdot str_1, r_1 + i_1 \cdot str_1)$
  for $i_2 \leftarrow I_2$ **to** $I_2 + m_2 - 1$
  do
    $L_2 = (l_2 + i_2 \cdot str_2, r_2 + i_2 \cdot str_2)$
    if $(LI_{L_1 \bigcap L_2} \neq \emptyset)$
      then
        $l = LOW(LI_{L_1 \bigcap L_2})$
        $r = HIGH(LI_{L_1 \bigcap L_2})$
        $str = FP_{F_1 \bigcap F_2}$
        $num = \min(\left\lfloor \frac{num_1 - i_1 - 1}{m_1} \right\rfloor, \left\lfloor \frac{num_2 - i_2 - 1}{m_2} \right\rfloor) + 1$
        $FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup (l, r, str, num)$

Figure 5.6 FALLS Intersection Algorithm Based on Periods

us $str = 8$. Finally, the value of num is calculated to be 4, which gives us only a single FALLS $FI_{F_1 \bigcap F_2} = (2, 3, 8, 4)$ for the intersection.

In the example just considered, we had only one resultant FALLS. This may not be the case in general. Intuitively, we can see that there at most $m_1 + m_2$ distinct LOWs and $HIGH$s in a period given m_1 members for the first FALLS and m_2 members for the second FALLS; each of these can be the LOW or $HIGH$ for at most one intersection. Therefore, the maximum number of FALLS that can be produced using our intersection algorithm can be shown to be $m_1 + m_2$. In practice, m_1 and m_2 are small, which means few resultant FALLS during a FALLS intersection.

Although the algorithm outlined above substantially cuts down on the number of iterations performed as compared to the brute force algorithm described first, it is still not very efficient. This can be seen by considering the example of Figure 5.7. For this example, $m_1 = 1$ and $m_2 = 4$, which means our algorithm iterates four times. However, we see that the intersection consists of just one FALLS, which means three of the iterations produce no FALLS and are thus wasted. This gives us the possibility of constructing another algorithm which still looks at pairs of members within a period, but does not consider pairs that do not intersect. This pruning is

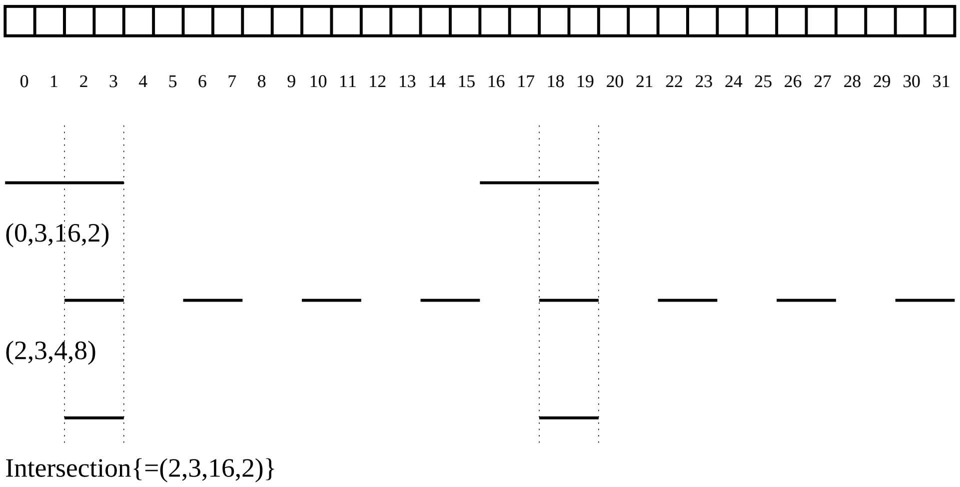


Figure 5.7 Another Example of FALLS Intersection

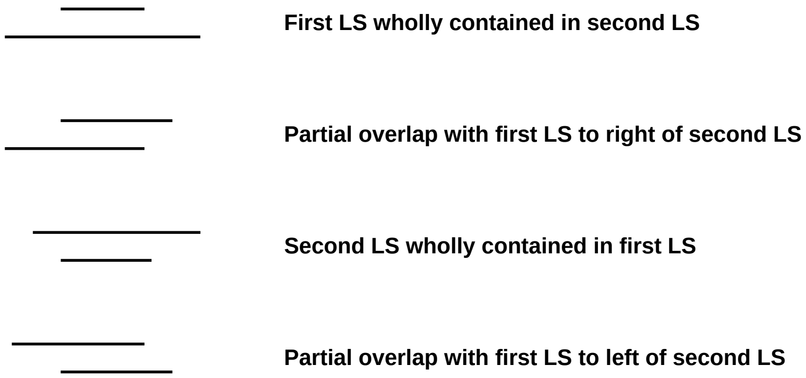


Figure 5.8 FALLS Intersection Conditions

done by considering the intersection of a pair of LS's $L_1 = (l_1, r_1)$ and $L_2 = (l_2, r_2)$. We see that this intersection is nonempty when $\max(l_1, l_2) \leq \min(r_1, r_2)$. This can happen under one of four conditions as listed below

$$
\begin{aligned}
& l_2 \leq l_1, \quad r_1 \leq r_2 \\
& l_2 \leq l_1, \quad l_1 \leq r_2, \quad r_2 \leq r_1 \\
& l_1 \leq l_2, \quad r_2 \leq r_1, \\
& l_1 \leq l_2, \quad l_2 \leq r_1, \quad r_1 \leq r_2
\end{aligned}
$$

Figure 5.8 illustrates the four cases.

Applying these conditions to determine when a pair of members (i_1, i_2) of two FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$ will intersect gives us

$$
\begin{array}{lll}
l_2 + i_2 \cdot str_2 \leq l_1 + i_1 \cdot str_1, & r_1 + i_1 \cdot str_1 \leq r_2 + i_2 \cdot str_2 & \\
l_2 + i_2 \cdot str_2 \leq l_1 + i_1 \cdot str_1, & l_1 + i_1 \cdot str_1 \leq r_2 + i_2 \cdot str_2, & r_2 + i_2 \cdot str_2 \leq r_1 + i_1 \cdot str_1 \\
l_1 + i_1 \cdot str_1 \leq l_2 + i_2 \cdot str_2, & r_2 + i_2 \cdot str_2 \leq r_1 + i_1 \cdot str_1 & \\
l_1 + i_1 \cdot str_1 \leq l_2 + i_2 \cdot str_2, & l_2 + i_2 \cdot str_2 \leq r_1 + i_1 \cdot str_1, & r_1 + i_1 \cdot str_1 \leq r_2 + i_2 \cdot str_2
\end{array}
$$

By performing some straightforward algebraic manipulations, we can reduce these conditions to the following

$$
\begin{array}{l}
i_2 \geq i_1 \cdot \frac{str_1}{str_2} + \frac{l_1 - r_2}{str_2} \\
i_2 \leq i_1 \cdot \frac{str_1}{str_2} + \frac{r_1 - l_2}{str_2}
\end{array}
\tag{5.3}
$$

The inequalities above give us a method to determine those members of the two FALLS which will actually intersect. It can be seen that given a member of the first FALLS, we can use these conditions to determine the lower and upper bounds of members of the other FALLS (i_2s) that will intersect with the given member. We can now construct an efficient intersection algorithm as shown in Figure 5.9.

Note that we do not check for an empty $LI_{L_1 \bigcap L_2}$ because we are guaranteed it is non-empty by iterating over the loop bounds computed using the conditions listed above. For our example of Figure 5.7, we can see that our algorithm will iterate only once and produce the FALLS $FI_{F_1 \bigcap F_2} = (2, 3, 16, 2)$.

It can be seen that the inequalities of Equation (5.3) describe the region between a pair of parallel lines in the (i_1, i_2) space. The slope for the lines is $\frac{str_1}{str_2}$ and their intercepts are $\frac{l_1 - r_2}{str_2}$ and $\frac{r_1 - l_2}{str_2}$.

5.2.2.1 Constant time check for FALLS intersection

To better understand the FALLS intersection process, consider the grid of size $num_1 \times num_2$ shown in Figure 5.10. Each point in this grid represents a pair of members of the given pair of FALLS. Using each of the intersection algorithms described results in the examination of a certain number of points in this grid. Under the brute force approach, we examine every point in the grid (white, grey and black points). The intersection based on periods considers only

EFFICIENT()

$FI_{F_1 \bigcap F_2} = \emptyset$

$I_1 = \max(0, \left\lceil \frac{l_2 - r_1}{str_1} \right\rceil)$

$I_2 = \max(0, \left\lceil \frac{l_1 - r_2}{str_2} \right\rceil)$

for $i_1 \leftarrow I_1$ **to** $\min(I_1 + m_1 - 1, num_1 - 1)$

do

  $L_1 = (l_1 + i_1 \cdot str_1, r_1 + i_1 \cdot str_1)$

  for $i_2 \leftarrow \max(I_2, \left\lceil \frac{i_1 \cdot str_1 + l_1 - r_2}{str_2} \right\rceil)$ **to** $\min(\left\lfloor \frac{i_1 \cdot str_1 + r_1 - l_2}{str_2} \right\rfloor, I_2 + m_2 - 1, num_2 - 1)$

  do

    $L_2 = (l_2 + i_2 \cdot str_2, r_2 + i_2 \cdot str_2)$

    $l = LOW(LI_{L_1 \bigcap L_2})$

    $r = HIGH(LI_{L_1 \bigcap L_2})$

    $str = FP_{F_1 \bigcap F_2}$

    $num = \min(\frac{num_1 - i_1 - 1}{m_1}, \frac{num_2 - i_2 - 1}{m_2}) + 1$

    $FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup (l, r, str, num)$

Figure 5.9 Efficient FALLS Intersection Algorithm Based on Periods

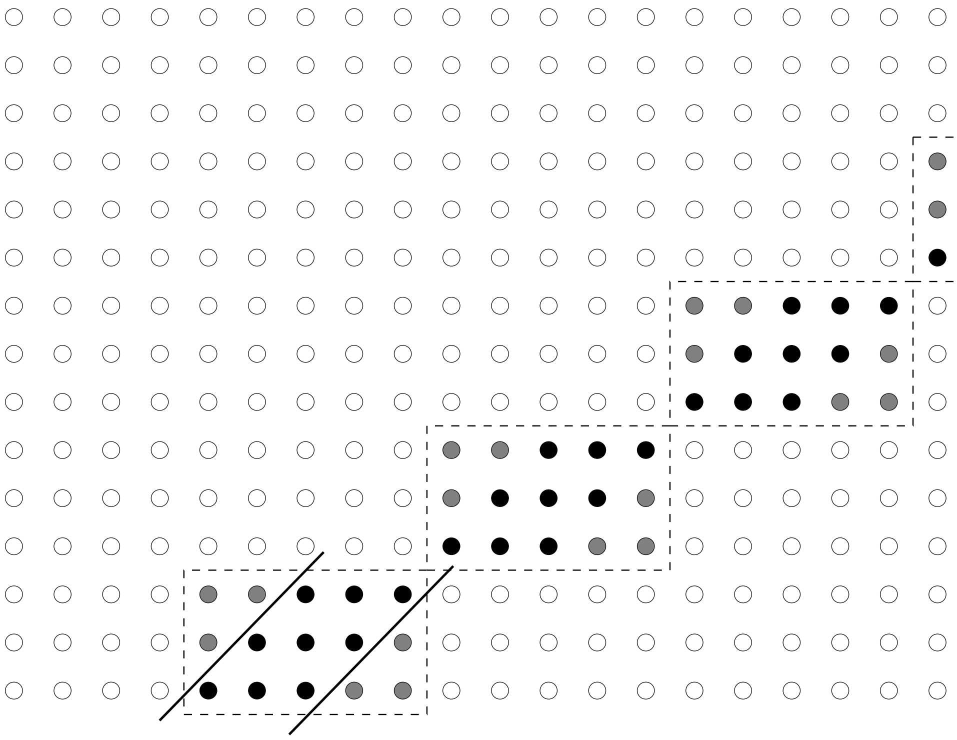


Figure 5.10 FALLS Intersection Summary

the points inside a box of dimensions $m_1 \times m_2$ (grey and black points); the lower left corner of the box is located at (I_1, I_2). Finally, the efficient intersection based on periods draws a pair of parallel lines (Equation (5.3)) in this grid and considers only those points in the box of size $m_1 \times m_2$ that lie on either line or between the lines (black points).

From Figure 5.10 we can see that a given pair of FALLS will not intersect if there are no integer points (i_1, i_2) that are in a box of size $m_1 \times m_2$ (lower-left corner at (I_1, I_2)) and lie on or between the pair of parallel lines given by Equation (5.3). We can simplify the problem slightly using the transformations $i_1' = i_1 - I_1$ and $i_2' = i_2 - I_2$ in Equation (5.3)

$$\begin{aligned} i_2' &\geq i_1' \cdot \tfrac{str_1}{str_2} + \tfrac{l_1 - r_2 - I_2 \cdot str_2 + I_1 \cdot str_1}{str_2} \\ i_2' &\leq i_1' \cdot \tfrac{str_1}{str_2} + \tfrac{r_1 - l_2 - I_2 \cdot str_2 + I_1 \cdot str_1}{str_2} \end{aligned} \tag{5.4}$$

Our check for a nonempty FALLS intersection now involves checking if there are any points inside a box of size $m_1 \times m_2$ (lower left corner at origin) that also lie on or between a pair of parallel lines given by Equation (5.4)

$$\begin{aligned} i_2' &\geq i_1' \cdot \tfrac{str_2}{str_1} + \tfrac{l_1 - r_2 - I_2 \cdot str_2 + I_1 \cdot str_1}{str_2} \\ i_2' &\leq i_1' \cdot \tfrac{str_2}{str_1} + \tfrac{r_1 - l_2 - I_2 \cdot str_2 + I_1 \cdot str_1}{str_2} \\ 0 &\leq i_1' \leq m_1 - 1 \\ 0 &\leq i_2' \leq m_2 - 1 \end{aligned} \tag{5.5}$$

We will now proceed to derive a constant time check for nonempty FALLS intersection. If the check passes, we can use the efficient algorithm based on periods to compute the actual intersection; if the check fails, we need to go no further.

We use the Fourier elimination method [77] to solve the set of inequalities in Equation (5.5). Using the Fourier method involves selecting an order for elimination of variables; the last variable in this list will have constant bounds associated with it. In our case, if we were to select the order $\{i_2', i_1'\}$, we would obtain constant bounds for i_1' that satisfy all the inequalities in Equation (5.5). On the other hand, if we select the elimination order to be $\{i_1', i_2'\}$, we would obtain constant bounds for i_2' that satisfy all the inequalities in Equation (5.5). The intuition behind our constant check is to use both orders of elimination listed above and obtain bounds on i_1' and i_2'; if either variable has a lower bound that is greater than the upper bound, there is

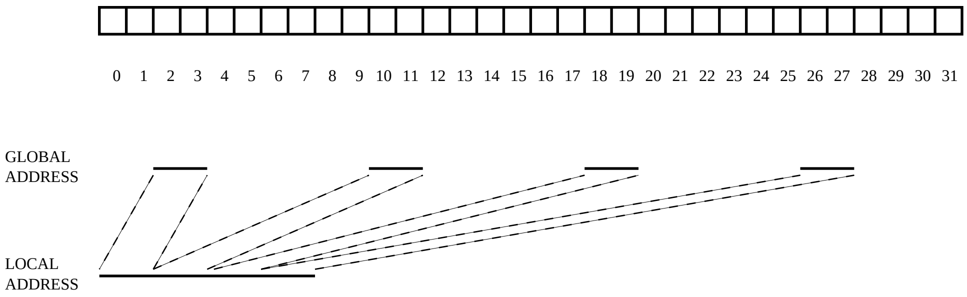


Figure 5.11 Generating Local Addresses Using FALLS

no point $(i_1^{'}, i_2^{'})$ that satisfies the inequalities in Equation (5.5). Using the Fourier method, we obtain bounds on $i_1^{'}$ and $i_2^{'}$ that are given by

$$\begin{array}{l} \max(0, \left\lceil \frac{l_2 - r_1 + I_2 \cdot str_2 - I_1 \cdot str_1}{str_1} \right\rceil) \leq i_1^{'} \leq \min(m_1 - 1, \left\lfloor \frac{r_2 - l_1 + I_2 \cdot str_2 - I_1 \cdot str_1 + (m_2 - 1) \cdot str_2}{str_1} \right\rfloor) \\ \max(0, \left\lceil \frac{l_1 - r_2 - I_2 \cdot str_2 + I_1 \cdot str_1}{str_2} \right\rceil) \leq i_2^{'} \leq \min(m_2 - 1, \left\lfloor \frac{r_1 - l_2 - I_2 \cdot str_2 + I_1 \cdot str_1 + (m_1 - 1) \cdot str_1}{str_2} \right\rfloor) \end{array} \tag{5.6}$$

Our constant time Boolean check $B_{F_1 \bigcap F_2}$ uses the parameters of the two FALLS to compute the values of the bound expressions given in Equation (5.6) and ensures that the value of the lower bound for $i_1^{'}$ and $i_2^{'}$ does not exceed the value of their upper bound. Note that the Fourier Elimination method was only used to compute the expressions for these bounds; it is not used during actual program execution.

5.2.2.2 Local addresses

We now discuss the issues related to generating local addresses for accessing elements of a distributed array during redistribution. Chatterjee et al. [78] have discussed a state machine approach for generating local addresses for affine accesses to data distributed in a `BLOCKCYCLIC` manner. While this work is more general than the approach we present below, our approach is simple and adequate for the data redistribution problem we are considering.

The FALLS representation is particularly useful for generating local addresses since the index of a member in a FALLS for a processor can be used to determine the local address for elements of the member. Figure 5.11 shows the arrangement in local memory for the blocks of elements that $P1$ owns under a `BLOCKCYCLIC(2)` distribution of a 32 element array on four

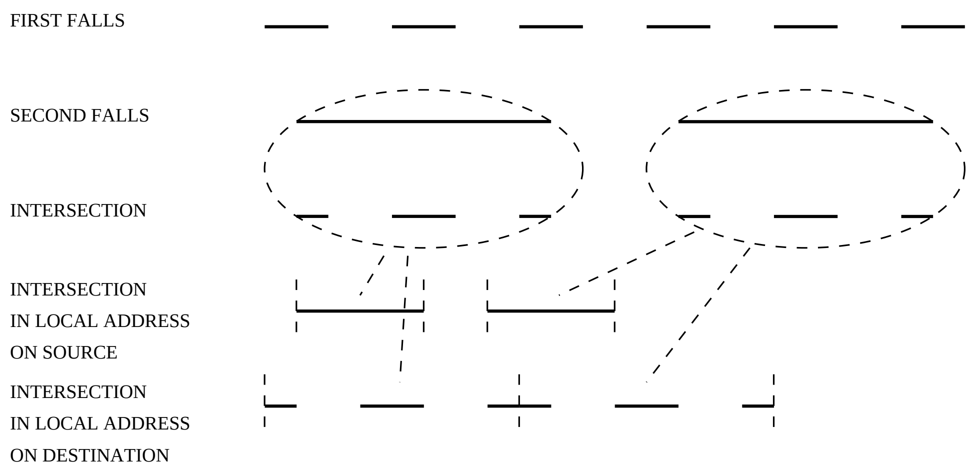


Figure 5.12 Generating FALLS with Good Memory Locality

processors with indices $(P0 - P3)$. To compute the local address L of the element which has global address j and is contained in the ith member of a FALLS $(F = l, r, str, num)$, we use

$$L = i * (r - l + 1) + j \bmod (r - l + 1)$$

From the intersection algorithm described in Figure 5.9, we see that the indices of the members of the two FALLS that are involved in the intersection are available to us (i_1, i_2). This means we use i_1 to compute the local address of all the elements of the resulting intersection on the sending processor; similarly, we can use i_2 to compute local addresses at the receiving processor. For example, if we consider the intersection shown in Figure 5.7, the local address (L_1) for the first element of the first resulting segment (global address $j = 2$) on the sending processor $(i_1 = 0)$ can be computed as

$$L_1 = 0 * (3 - 0 + 1) + 2 \bmod (3 - 0 + 1) \Rightarrow L_1 = 2$$

In a similar manner we can compute the local address (L_2) for the element on the receiving processor $(i_2 = 0)$

$$L_2 = 0 * (3 - 2 + 1) + 2 \bmod (3 - 2 + 1) \Rightarrow L_2 = 0$$

5.2.2.3 Improving memory locality

As discussed before, we generate a set of FALLS during the computation of the intersection of a given pair of FALLS. For redistribution, we have to pack all the elements of each of the generated FALLS into a contiguous buffer for transfer from the source processor to the destination processor where the elements have to be unpacked from the buffer. To make this process more efficient, during FALLS intersection, we generate FALLS with individual members being as large as possible. Doing this results in maximizing unit-stride accesses and, therefore, better utilization of the memory hierarchy.

To intuitively understand our method for generating FALLS with large individual members during FALLS intersection, consider Figure 5.12. From this figure, we see that there are multiple members of the first FALLS that intersect with a single member of the second FALLS. Under our algorithm of Figure 5.9, we generate a separate FALLS for each member of the first FALLS. However, as shown in the figure, all the members of the first FALLS are contiguously laid out in the local memory of the source processor. This means we can generate a single FALLS for all members of the first FALLS by increasing the size of individual members. On the other hand, we cannot avoid strided accesses on the destination processor, we generate a single FALLS whose members themselves are FALLS.

5.2.3 Redistribution algorithm

Our redistribution algorithm for a single dimensional array is shown in Figure 5.13; Figure 5.13(a) shows the part of the algorithm executed on the source processors and Figure 5.13(b) is the part executed by the destination processors. The various quantities used in the algorithm are

- $proc_1$ - number of source processors
- $dist_1$ - source distribution descriptor
- $proc_2$ - number of destination processors
- $dist_2$ - destination distribution descriptor
- $getid()$ - obtains processor index for calling processor

REDISTRIBUTE-SEND()

$p_1 = getid()$
$F^{p_1} = getfalls(p_1, dist_1)$
for $p_2 \leftarrow 0$ **to** $proc_2 - 1$
do
    $F^{p_2} = getfalls(p_2, dist_2)$
    if $B_{F^{p_1} \bigcap F^{p_2}} =$ TRUE
        then
            $FI_{F^{p_1} \bigcap F^{p_2}} = interfalls(F^{p_1}, F^{p_2})$
            $packfalls(FI_{F^{p_1} \bigcap F^{p_2}}, buffer)$
            $send(p_2, buffer)$

(a) Procedure Used by Sending Processors

REDISTRIBUTE-RECV()

$p_2 = getid()$
$F^{p_2} = getfalls(p_2, dist_2)$
for $p_1 \leftarrow 0$ **to** $proc_1 - 1$
do
    $F^{p_1} = getfalls(p_1, dist_1)$
    if $B_{F^{p_1} \bigcap F^{p_2}} =$ TRUE
        then
            $FI_{F^{p_1} \bigcap F^{p_2}} = interfalls(F^{p_1}, F^{p_2})$
            $recv(p_1, buffer)$
            $unpackfalls(FI_{F^{p_1} \bigcap F^{p_2}}, buffer)$

(b) Procedure Used by Receiving Processors

Figure 5.13 Redistribution Algorithm

- $getfalls(p, dist)$ - obtains FALLS for processor p under distribution $dist$
- $interfalls(F_1, F_2)$ - obtains the FALLS intersection for the pair of FALLS F_1 and F_2
- $packfalls(FI, buffer)$ - packs the elements represented by a set of FALLS FI into a contiguous buffer
- $unpackfalls(FI, buffer)$ - unpacks the elements represented by a set of FALLS FI from a contiguous buffer

5.2.3.1 Communication scheduling

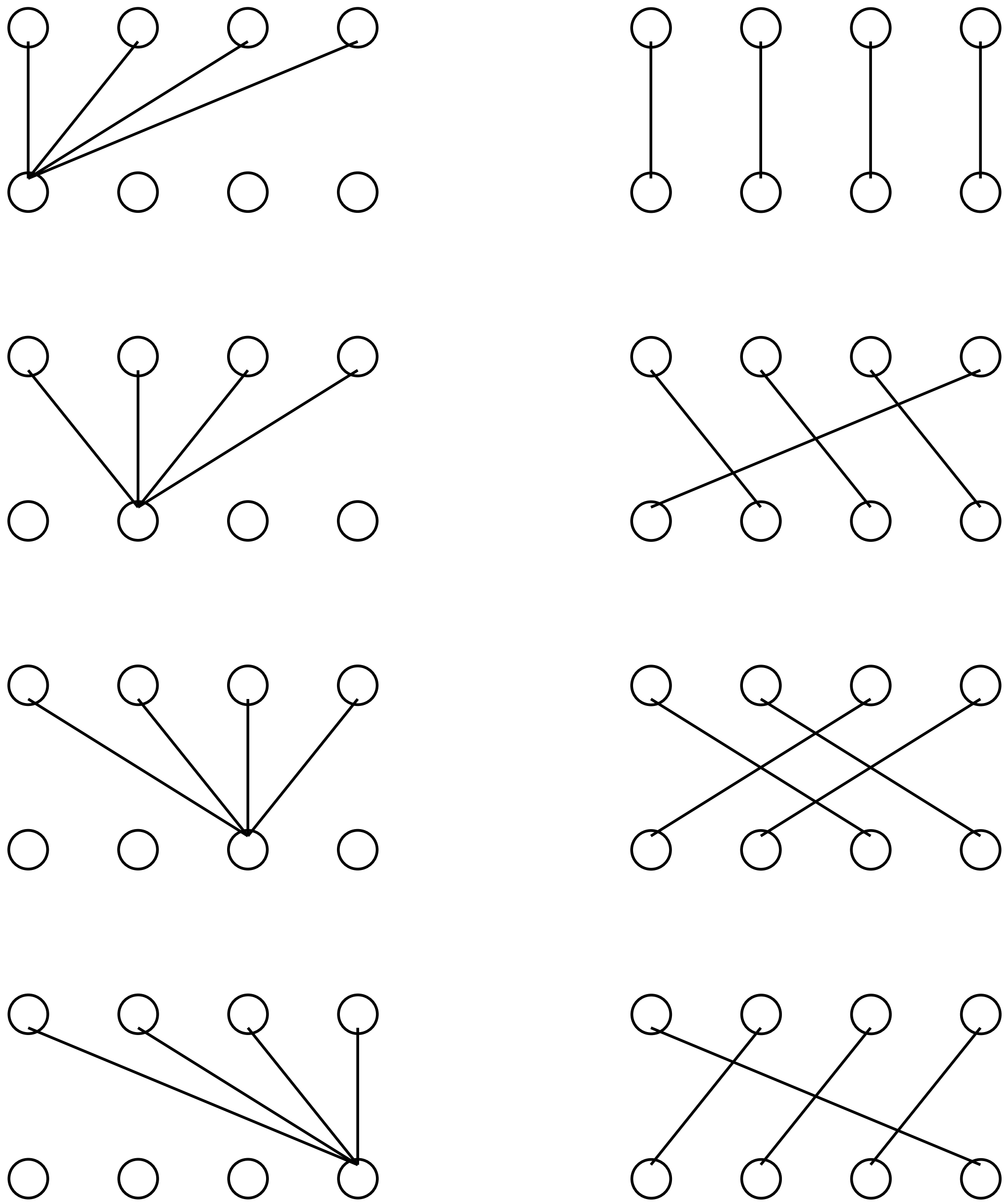
Figure 5.14(a) shows the sequence of events that can occur during a redistribution involving four source and four destination processors using the algorithm of Figure 5.13. We see that each destination processor receives all of its messages simultaneously; this may lead to communication bottlenecks. We can use simple schemes to avoid such bottlenecks by having the communication distributed more evenly; the effect of such schemes is illustrated in Figure 5.14(b). One of the schemes we use is to have each source processor start with a different destination processor index, the modified algorithm implementing this scheme is shown in Figure 5.15.

5.2.4 Multidimensional array redistribution

So far, we have only considered a one-dimensional array for describing the FALLS representation, the FALLS intersection algorithm and the redistribution algorithm based on the FALLS intersection algorithm.

For multidimensional arrays, the FALLS representation for the elements owned by each processor is constructed independently for each array dimension. To illustrate this, we have provided examples in Figure 5.16. This figure shows the FALLS representation for a processor with coordinates $(1, 0)$ under two different distribution schemes for a two-dimensional array on a 4×4 processor grid. The first scheme has the array distributed in a `(BLOCK,CYCLIC)` manner and the second scheme has the array distributed in a `(CYCLIC,BLOCKCYCLIC(4))` manner.

The multidimensional FALLS intersection algorithm shown in Figure 5.17 treats each dimension of the array independently and uses the one-dimensional FALLS intersection algorithm. After performing the dimension-by-dimension intersection, we can see that the set of resulting FALLS represents the set of indices in each array dimension that have to be used for transferring data elements during redistribution. Thus, during redistribution, we construct rectangular sections of elements by combining the indices for all dimensions. This is made clear in the example of a two-dimensional FALLS intersection shown in Figure 5.18. The source and target distributions for this redistribution are assumed to be the ones shown in Figure 5.16. As we can see, the intersection along the row dimension indicates that all elements of row 0 are to be sent from source processor 1 to target processor 1. The intersection along the column dimension indicates that all elements in columns $5, 7, 13, 15, 21, 23, 29, 31$ are to be



(a) Without Scheduling (a) With Scheduling

Figure 5.14 Communication Patterns During Redistribution

SCHEDULED-SEND()

$p_1 = getid()$
$F^{p_1} = getfalls(p_1, dist_1)$
for $p_2^{'} \leftarrow 0$ **to** $proc_2 - 1$
do
    $p_2 = (p_1 + p_2^{'}) \bmod proc_2$
    $F^{p_2} = getfalls(p_2, dist_2)$
    if $B_{F^{p_1} \bigcap F^{p_2}} =$ TRUE
        then
            $FI_{F^{p_1} \bigcap F^{p_2}} = interfalls(F^{p_1}, F^{p_2})$
            $packfalls(FI_{F^{p_1} \bigcap F^{p_2}}, buffer)$
            $send(p_2, buffer)$

(a) Procedure Used by Sending Processors

SCHEDULED-RECV()

$p_2 = getid()$
$F^{p_2} = getfalls(p_2, dist_2)$
for $p_1^{'} \leftarrow 0$ **to** $proc_1 - 1$
do
    $p_1 = (p_2 + p_1^{'}) \bmod proc_1$
    $F^{p_1} = getfalls(p_1, dist_1)$
    if $B_{F^{p_1} \bigcap F^{p_2}} =$ TRUE
        then
            $FI_{F^{p_1} \bigcap F^{p_2}} = interfalls(F^{p_1}, F^{p_2})$
            $recv(p_1, buffer)$
            $unpackfalls(FI_{F^{p_1} \bigcap F^{p_2}}, buffer)$

(b) Procedure Used by Receiving Processors

Figure 5.15 Redistribution Algorithm with Communication Scheduling

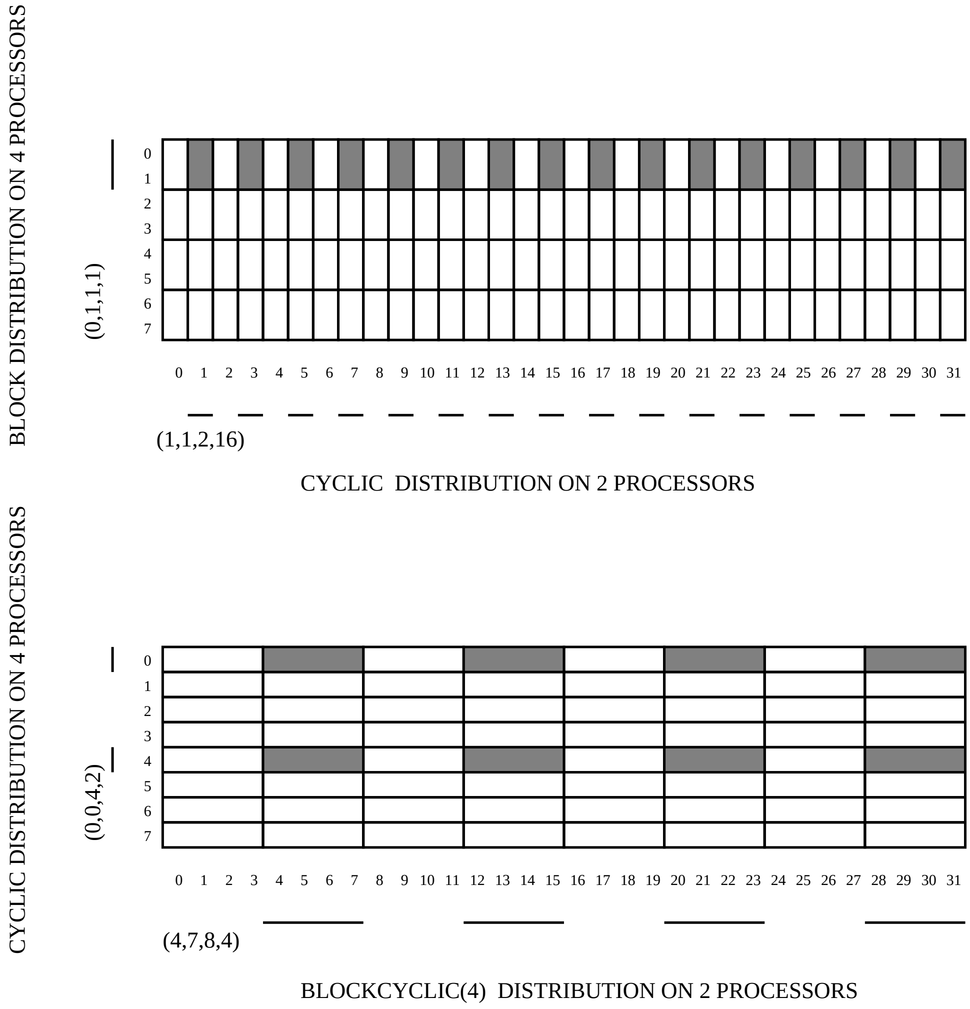


Figure 5.16 Examples of FALLS for Multidimensional Arrays

MULTIDIMENSION()

for $dim \leftarrow 0$ **to** $NumDims$
do
    $FI^{dim}_{F_1^{dim} F_2^{dim}} = \emptyset$
    $I_1^{dim} = \max(0, \left\lceil \frac{l_2^{dim} - r_1^{dim}}{str_1^{dim}} \right\rceil)$
    for $i_1 \leftarrow I_1^{dim}$ **to** $\min(I_1^{dim} + m_1^{dim} - 1, num_1^{dim} - 1)$
    do
        $L_1 = (l_1^{dim} + i_1 \cdot str_1^{dim}, r_1^{dim} + i_1 \cdot str_1^{dim})$
        for $i_2 \leftarrow \max(0, \left\lceil \frac{l_1^{dim} - r_2^{dim}}{str_2^{dim}} \right\rceil)$ **to** $\min(\left\lfloor \frac{r_1^{dim} - l_2^{dim}}{str_2^{dim}} \right\rfloor, m_2^{dim} - 1, num_2^{dim} - 1)$
        do
            $L_2 = (l_2^{dim} + i_2 \cdot str_2^{dim}, r_2^{dim} + i_2 \cdot str_2^{dim})$
            $l = LOW(LI_{L_1 L_2})$
            $r = HIGH(LI_{L_1 L_2})$
            $str = FP^{dim}_{F_1^{dim} F_2^{dim}}$
            $num = \min(\frac{num_1^{dim} - i_1 - 1}{m_1^{dim}}, \frac{num_2^{dim} - i_2 - 1}{m_2^{dim}}) + 1$
            $FI_{F_1^{dim} F_2^{dim}} = FI_{F_1^{dim} F_2^{dim}} \bigcup (l, r, str, num)$

Figure 5.17 Multidimensional FALLS Intersection Algorithm

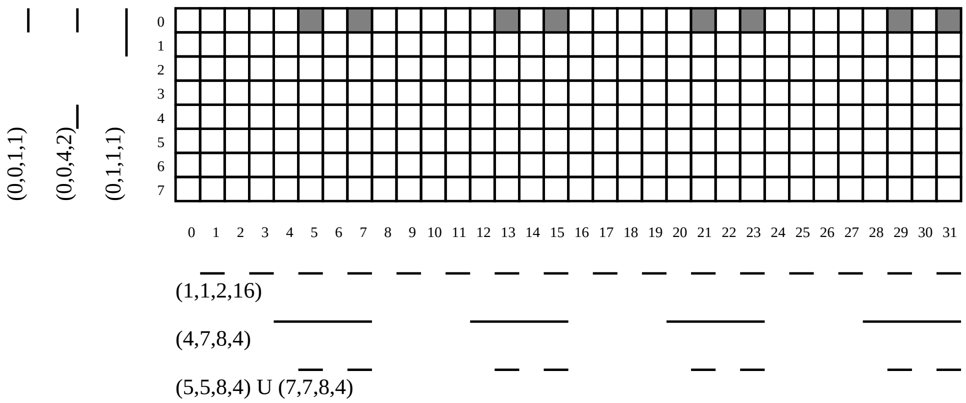


Figure 5.18 Example of Multidimensional FALLS Intersection

sent between these processors. Combining the two sets of indices gives us the set of elements $(0,5), (0,7), (0,13), (0,15), (0,21), (0,23), (0,29), (0,31)$. We indicate these by the shaded areas.

Our multidimensional intersection algorithm scales linearly with the number of dimensions involved. This is a significant advantage over the methods of [63].

5.2.5 Multiarray redistribution

For multiple arrays being redistributed from one processor set to another processor set, we pack all the data to be transferred between a pair of processors for all the arrays into a single buffer before sending its contents. In this way, we ensure that no more than one message is sent between processors even though they may communicate data for more than one array. The advantage is that we have one long message instead of multiple short messages.

5.3 Implementation and Results

The redistribution algorithm based on the FALLS representation has been implemented as a pair of library routines called *GroupSend* and *GroupRecv*. These functions require information about the source and target processor grids and information about each of the arrays being redistributed. The *GroupSend* function is executed by each processor in the source grid. Each processor analyzes the communication requirements for all the arrays being redistributed and assembles a single message for each target processor that needs any data from it. The *GroupRecv* function essentially does the reverse.

We present below experimental results that demonstrate the usefulness of the optimizations we described earlier – memory locality improvement and communication scheduling. Later, we present evidence to demonstrate the scalability of our redistribution algorithms.

To measure the performance improvements provided by our optimizations, we generated and timed (in milliseconds) our basic unoptimized algorithm (*Unopt*), a modified version that implements our memory locality optimization (*Local*) and a modified version of *Local* that implements the communication scheduling optimization (*Sched*). We timed a total of 27 redistributions using all possible combinations of

(1) Three source-target distribution pairs:

   (a) `BLOCKCYCLIC(3),BLOCK` to `CYCLIC,BLOCKCYCLIC(5)`

   (b) `BLOCKCYCLIC(3),BLOCKCYCLIC(7)` to `BLOCKCYCLIC(5),CYCLIC`

   (c) `BLOCK,ALL` to `ALL,BLOCK`

(2) Three arrays sizes:

Table 5.1 `BLOCKCYCLIC(3),BLOCK` to `CYCLIC,BLOCKCYCLIC(5)` Redistribution Time (in milliseconds) on the CM-5

Size	P_s	P_t	$Unopt$(ms)	$Local$(ms)	$Sched$(ms)
100×100	4×4	3×5	12.5	10.4	7.9
	2×6	3×3	11.1	10.2	9.8
	3×5	4×3	13.5	11.1	8.8
200×200	4×4	3×5	23.2	19.6	16.0
	2×6	3×3	30.2	27.9	28.1
	3×5	4×3	28.2	23.2	20.3
400×400	4×4	3×5	72.0	68.7	55.6
	2×6	3×3	99.5	89.5	97.2
	3×5	4×3	84.1	79.6	65.4

Table 5.2 `BLOCKCYCLIC(3),BLOCKCYCLIC(7)` to `BLOCKCYCLIC(5),CYCLIC` Redistribution Time (in milliseconds) on the CM-5

Size	P_s	P_t	$Unopt$(ms)	$Local$(ms)	$Sched$(ms)
100×100	5×2	4×3	13.0	11.5	9.5
	3×6	5×2	17.7	13.4	11.8
	4×5	3×3	19.1	14.0	11.5
200×200	5×2	4×3	31.6	30.0	24.7
	3×6	5×2	39.6	31.9	28.7
	4×5	3×3	33.9	27.4	22.3
400×400	5×2	4×3	86.0	95.9	73.7
	3×6	5×2	97.7	93.9	83.0
	4×5	3×3	82.5	89.7	65.8

(a) 100×100

(b) 200×200

(c) 400×400

(3) Three processor grids chosen independently for each distribution pair (shown along with the timings in Tables 5.1-5.3 and Tables 5.4-5.6).

The distributions and processor grids were chosen to show that our method works well across a range of regular redistributions on arbitrary processor sets. The results of our study are tabulated in Tables 5.1-5.3 for the Thinking Machines CM-5 and Tables 5.4-5.6 for the Intel Paragon. From these tables, we can make the following observations:

Table 5.3 `CYCLIC,ALL` to `ALL,CYCLIC` Redistribution Time (in milliseconds) on the CM-5

Size	P_s	P_t	$Unopt$(ms)	$Local$(ms)	$Sched$(ms)
100×100	8×1	1×16	25.4	10.0	9.8
	16×1	1×16	23.5	12.4	12.0
	10×1	1×16	23.6	10.6	10.1
200×200	8×1	1×16	81.2	25.6	25.2
	16×1	1×16	63.5	21.4	21.0
	10×1	1×16	76.3	24.4	22.9
400×400	8×1	1×16	290.0	79.4	76.8
	16×1	1×16	224.5	65.0	69.3
	10×1	1×16	256.1	78.6	73.5

Table 5.4 `BLOCKCYCLIC(3),BLOCK` to `CYCLIC,BLOCKCYCLIC(5)` Redistribution Time (in milliseconds) on the Paragon

Size	P_s	P_t	$Unopt$(ms)	$Local$(ms)	$Sched$(ms)
100×100	4×4	3×5	5.0	4.4	3.7
	2×6	3×3	4.8	4.6	4.0
	3×5	4×3	6.3	5.2	4.6
200×200	4×4	3×5	10.9	9.3	7.9
	2×6	3×3	13.3	12.4	10.7
	3×5	4×3	13.8	11.5	10.2
400×400	4×4	3×5	33.9	29.2	24.6
	2×6	3×3	47.1	42.5	40.1
	3×5	4×3	43.3	37.8	33.7

Table 5.5 `BLOCKCYCLIC(3),BLOCKCYCLIC(7)` to `BLOCKCYCLIC(5),CYCLIC` Redistribution Time (in milliseconds) on the Paragon

Size	P_s	P_t	$Unopt$(ms)	$Local$(ms)	$Sched$(ms)
100×100	5×2	4×3	6.1	5.2	5.3
	3×6	5×2	7.7	6.2	5.6
	4×5	3×3	8.9	6.2	5.8
200×200	5×2	4×3	13.1	12.0	12.0
	3×6	5×2	18.6	15.5	13.6
	4×5	3×3	15.1	12.0	10.5
400×400	5×2	4×3	41.2	40.0	38.9
	3×6	5×2	47.8	42.1	35.7
	4×5	3×3	37.6	33.6	28.2

Table 5.6 `CYCLIC,ALL` to `ALL,CYCLIC` Redistribution Time (in milliseconds) on the Paragon

Size	P_s	P_t	$Unopt$(ms)	$Local$(ms)	$Sched$(ms)
100×100	8×1	1×16	15.4	4.8	4.7
	16×1	1×16	12.8	5.3	5.4
	10×1	1×16	14.2	4.7	4.8
200×200	8×1	1×16	50.6	9.6	9.7
	16×1	1×16	37.1	9.5	9.6
	10×1	1×16	45.3	9.6	9.5
400×400	8×1	1×16	190.3	31.3	31.8
	16×1	1×16	133.0	27.6	28.3
	10×1	1×16	167.7	29.1	29.2

Table 5.7 Overhead for `BLOCKCYCLIC(3),BLOCK` to `CYCLIC,BLOCKCYCLIC(5)` Redistribution Communication Analysis on the CM-5

P_s	P_t	Size	Overhead for $Local$(ms)
4×4	3×5	100×100	1.77
		200×200	1.84
		400×400	1.81
3×5	4×3	100×100	2.25
		200×200	2.39
		400×400	2.38
2×6	3×3	100×100	1.80
		200×200	1.89
		400×400	1.86

- Both of our optimizations provide significant performance benefits on both target machines.
- The performance improvement becomes more appreciable as the array size increases. This means it is vital to use the optimizations for redistributions of large arrays.

We have tabulated the overheads of communication analysis for the *Local* algorithm for a `BLOCKCYCLIC(3),BLOCK` to `CYCLIC,BLOCKCYCLIC(5)` redistribution on both our target machines in Tables 5.7 and 5.8. From these tables, we can see that the redistribution analysis overhead is very small and essentially independent of the array size. This makes our communication analysis algorithms extremely scalable.

Table 5.8 Overhead for `BLOCKCYCLIC(3),BLOCK` to `CYCLIC,BLOCKCYCLIC(5)` Redistribution Communication Analysis on the Paragon

P_s	P_t	Size	Overhead for *Local*(ms)
4×4	3×5	100×100	0.97
		200×200	0.94
		400×400	0.94
3×5	4×3	100×100	0.95
		200×200	0.94
		400×400	0.95
2×6	3×3	100×100	1.29
		200×200	1.28
		400×400	1.28

5.4 Summary

In this chapter we discussed the problem of data redistribution and our solution to the problem. The analysis of the communication required for a data redistribution is performed using the FALLS representation. The representation captures the regularity in data distribution schemes and makes communication analysis very efficient. This results is very low overheads for data redistribution.

Having looked at all the aspects involved in the simultaneous exploitation of task and data parallelism, we present experimental results in the next chapter to demonstrate the practicality and usefulness of our optimization.

CHAPTER 6

IMPLEMENTATION AND RESULTS

6.1 Introduction

In this chapter we discuss experimental evidence that demonstrates the effectiveness of our proposed optimization - the simultaneous exploitation of task and data parallelism. In Section 6.2 we present a brief overview of the implementation of our extended HPF and MATLAB compilers. In the next section we briefly describe each of the benchmark applications used for our experiments. Finally, to demonstrate the potential benefits of our optimization, in Section 6.4, we present speedup measurements obtained for our benchmark applications on the Thinking Machines CM-5 and Intel Paragon.

6.2 Compiler Implementation and Run-time Support

As mentioned earlier, we have implemented our proposed optimization as part of the PARADIGM compiler framework [3],[17] to support extended HPF applications and in a MATLAB compiler that is based on ScaLAPACK [5],[6]. To maximize the sharing of code modules between the two languages, we translate both extended HPF as well as MATLAB applications into a common parse tree format. Thereafter, all other modules are common to either language. This is illustrated in Figure 6.1. For extended HPF applications, we use the Parafrase-2 compiler [44],[66] as a front end. In the case of MATLAB applications, we have built a separate front end. Once the input application is in the common parse tree format, we use the algorithms described in Section 3.2.3 to extract the hierarchical MDG structure for the application and then use the algorithms described in Section 3.3.5 to fill in the cost parameters for nodes and edges in the MDG. Next, we use the allocation and scheduling algorithms described in Section 4.3 to decide on an execution scheme for the application. Finally, we generate MPMD code.

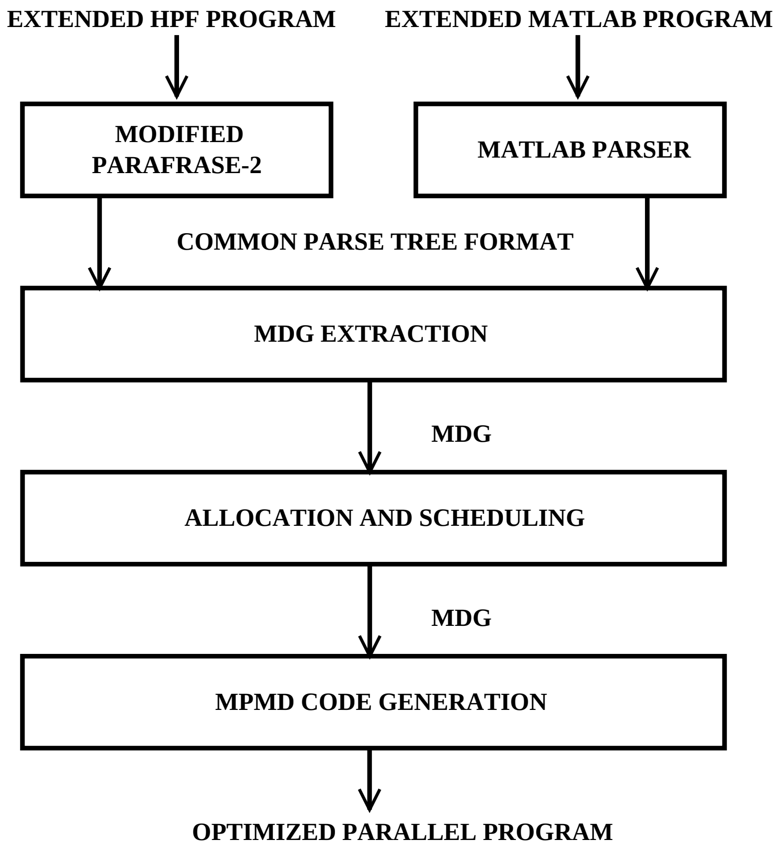


Figure 6.1 Implementation of the Compiler Modules

With respect to run-time support, the extended HPF and MATLAB compilers have different requirements. For the extended HPF compiler, our run-time support for task and data parallel execution and data redistribution is based on the MPI [73] message passing standard. Besides the growing popularity of MPI, other reasons for selecting MPI are

(1) The SPMD code generation module in the PARADIGM compiler [3],[17] supports the MPI library. Since data parallel code for individual tasks is generated by this module, we need to make our run-time routines compatible.

(2) MPI supports subpartition execution for program modules through the use of *Groups*, *Contexts* and *Communicators*. Groups bind processors together in a subpartition and contexts create a unique set of message IDs for the processors in a group to communicate safely. A group and a context together form a communicator.

Run-time support for the MATLAB compiler is based on the BLACS [40] message passing library. As mentioned before, ScaLAPACK routines correspond to data parallel code for individual tasks in MATLAB applications. Since ScaLAPACK is based on the BLACS library, we have to ensure compatibility of our run-time routines. The BLACS library supports subpartition execution for program modules through the use of *Process Grids* and *Contexts*. Process grids bind processors together in a subpartition and contexts create a unique set of message IDs for communication between processors in a process grid.

6.3 Benchmarks

As discussed in Chapter 3, we have limited the scope of HPF and MATLAB language constructs for the purpose of our research. This was done to facilitate rapid prototype development and testing of our algorithms. However, the limited scope of language constructs restricts our choice of benchmark applications from the vast choice available. At this point, we were able to select only a set of five benchmark applications for our experiments. As we expand the scope of language constructs, we are confident of being able to handle larger benchmarks. The extended HPF and MATLAB codes for all our benchmarks are provided in Appendix B. The applications are

(1) Cmmul : This is the simplest of our applications and corresponds to the multiplication of a pair of complex matrices. The MDG for this application is a simple MDG.

(2) Strassen : This application corresponds to the multiplication of a pair of matrices using the Strassen's Matrix Multiplication algorithm [79].The MDG for this application is a simple MDG.

(3) Cfd : This benchmark corresponds to the core of a Fourier-Chebyshev spectral computational fluid dynamics algorithm [80]. The MDG for this application is a simple MDG.

(4) Poly : This benchmark corresponds to the multiplication of a sequence of polynomials using fast Fourier transforms [81]. The MDG for this application is hierarchical and contains an `L` node.

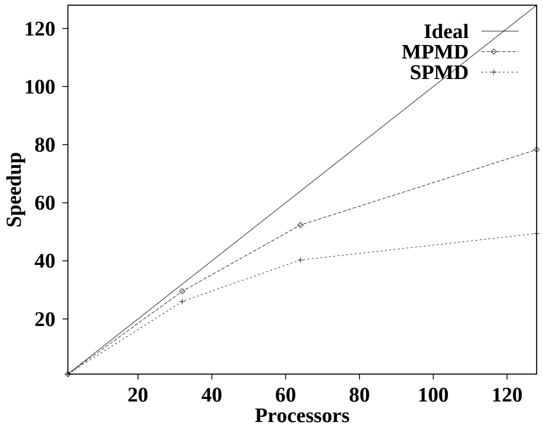
(5) Bicg : This application corresponds to the Biconjugate Gradient iterative method for solving systems of linear equations [65]. The MDG for this application is hierarchical and contains four `C` nodes and an `L` node.

6.4 Experiments

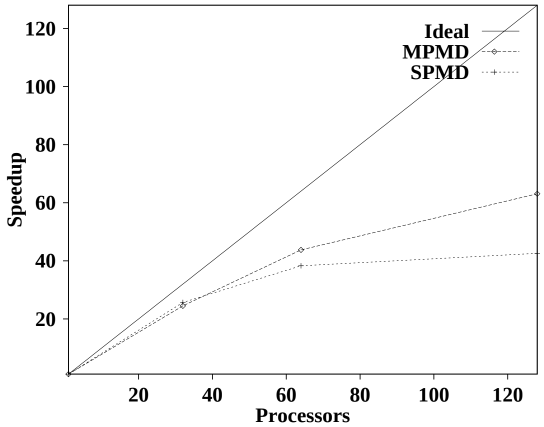
To evaluate the benefits of our proposed optimization of exploiting task and data parallelism, we compare the speedups obtained for our benchmarks using the optimization with speedups obtained using data parallelism alone. Figure 6.2 shows speedup data obtained for the Thinking Machines CM-5. Figure 6.3 shows speedup data obtained for the Intel Paragon. Speedups obtained using a task and data parallel scheme of execution are denoted by `MPMD`, and those obtained using a pure data parallel scheme of execution are denoted by `SPMD`. We used 32, 64 and 128 processors for our speedup measurements on the CM-5. On the other hand, we used 8, 16 and 32 processors for our measurements on the Paragon.

From the speedup data, we can make the following observations:

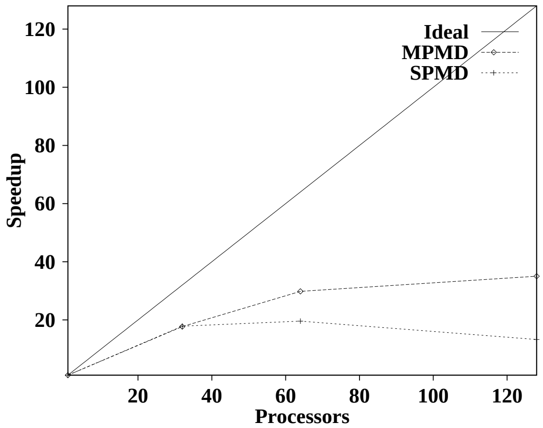
(1) Simultaneous exploitation of task and data parallelism provides increased speedups as compared to pure data parallelism in almost all cases. For some benchmarks, `MPMD` does slightly worse than `SPMD` for small system sizes but does much better for larger system sizes – there is a crossover point. The reason for this crossover in some benchmarks is



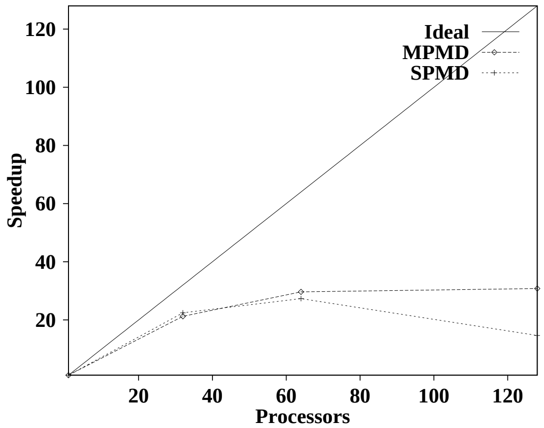
CMMUL



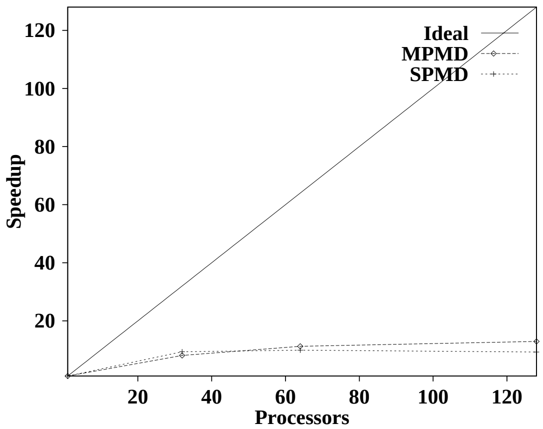
STRASSEN



CFD



POLY



BICG

Figure 6.2 Speedup Measurements for Benchmark Applications on the CM-5

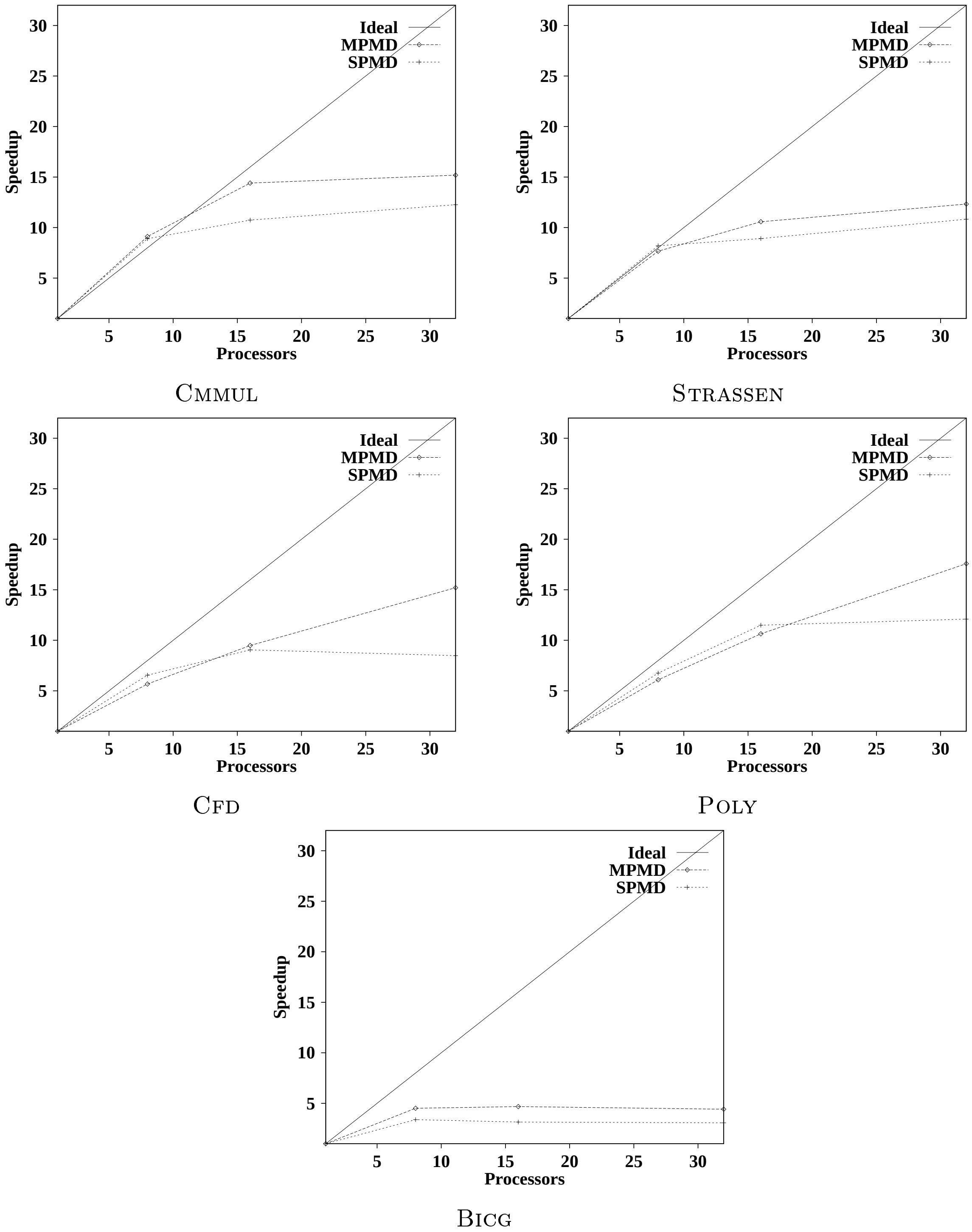


Figure 6.3 Speedup Measurements for Benchmark Applications on the Paragon

that the benefits obtained from executing tasks efficiently are outweighed by the cost of redistributing data between tasks.

(2) For many benchmarks, pure data parallelism actually produces lower speedups on larger system sizes. This makes the exploitation of task and data parallelism very critical.

(3) We use smaller system sizes on the Paragon as compared to the CM-5. Yet, we obtain significant speedup improvements on the Paragon when using task and data parallelism. The reason for this is the low computation to communication ratio on the Paragon, which results in data parallelism becoming rapidly inefficient, and, in turn leads to our optimization providing performance benefits at smaller system sizes as compared to the CM-5 where the computation to communication ratio is larger. Low computation to communication ratio is becoming a trend in newer distributed memory machines as processor speeds are growing faster than network speeds. This means that our optimization will provide significant performance benefits even for moderate data sizes and small system sizes.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this thesis, we have proposed a new compiler optimization for distributed memory machines - the simultaneous exploitation of task and data parallelism. Our optimization uses the small degree of task parallelism present in many scientific applications to control the degree of data parallelism used for individual tasks. We have implemented the optimization as part of an extended HPF compiler and a MATLAB compiler. As evidenced by our experimental results, the optimization can potentially provide significant performance gains for regular scientific applications.

In order to represent available task and data parallelism for an application, we use the Macro Dataflow Graph (MDG) representation. The MDG structure is hierarchical and supports complex program constructs such as loops and conditionals. We have developed algorithms to extract the MDG structure automatically from extended HPF and MATLAB applications. In addition, we have developed detailed cost models for nodes and edges in the MDG and proved some mathematical properties for these cost functions. These properties are exploited by our allocation and scheduling algorithm.

Given an MDG representation for an application, we use an allocation and scheduling approach to decide on a suitable task and data parallel execution scheme. Allocation decides on the number of processors to use for each data parallel task in the given application and scheduling decides on an order of execution for the tasks. The allocation and scheduling problem is NP-complete in the discrete domain. The key idea behind our approximate allocation and scheduling algorithm is to solve the problem optimally in polynomial time using a continuous domain formulation and then round off this solution for the discrete domain. This provides us with near optimal results in practice.

Data redistribution is critical for implementing a task and data parallel execution scheme; we have developed very efficient routines for this purpose. Our data redistribution analysis is based on a new mathematical representation for data distributions called FALLS. The FALLS representation captures information about regularity in data distributions and allows for effective use of this regularity.

There are some possible improvements to be made to our work. One of the issues would be to consider affinity-based scheduling techniques to eliminate data redistribution. It would also be interesting to consider run-time schemes to augment our static allocation and scheduling for large iterative applications. Our allocation and scheduling algorithm relies heavily on cost estimates; a run-time scheme could monitor the actual performance and fine tune the allocation and scheduling to compensate for any estimation errors.

Another important future direction would be to investigate techniques for exploiting task and data parallelism in irregular scientific applications. These applications would require dynamic run-time allocation and scheduling and data redistribution algorithms.

APPENDIX A

CONVEX AND POSYNOMIAL FUNCTIONS

In this appendix, we provide a brief overview of the theory of convex and posynomial functions. A detailed discussion of convex functions and convex programming can be found in Luenberger's book [82]. A discussion of the theory of posynomial functions is provided by Ecker in [83]. We have selected a few important and relevant points about these functions for our discussion here.

A.1 Convex Sets

A set C in $\mathbf{R}^n$ is said to be *convex* if, for every $\mathbf{x}_1$, $\mathbf{x}_2 \in C$, and every real number α, $0 \leq \alpha \leq 1$, the point $\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2 \in C$.

This definition can be interpreted geometrically as stating that a set is convex if, given two points in the set, every point on the line segment joining the two points is also a member of the set. Examples of convex and nonconvex sets are shown in Figure A.1.

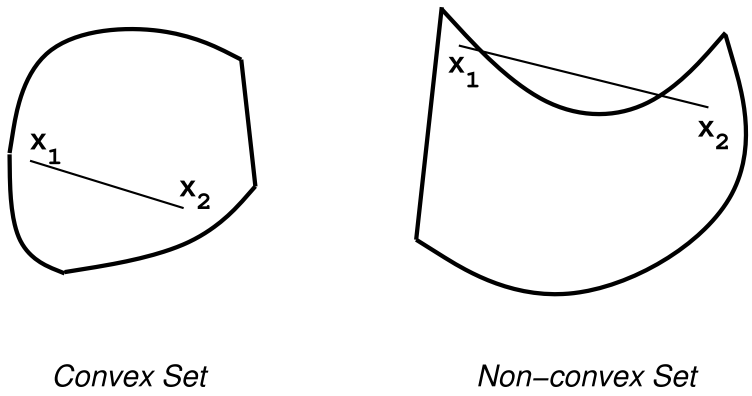


Figure A.1 Convex Sets

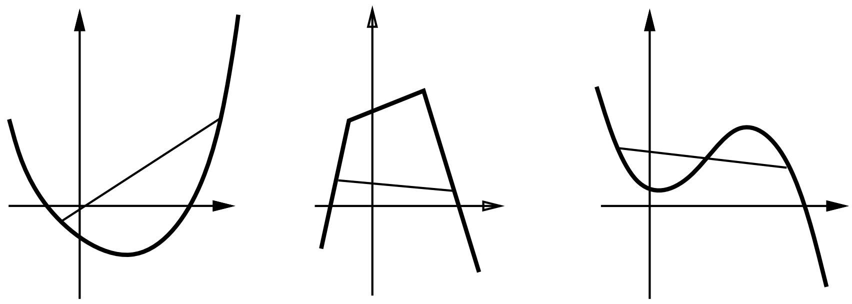


Figure A.2 Convex Functions

A.2 Convex Functions

Definition : A function f defined on a convex set Ω is said to be *convex* if, for every $\mathbf{x}_1$, $\mathbf{x}_2$ $\in \Omega$, and every α, $0 \leq \alpha \leq 1$,

$$f(\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2) \leq \alpha f(\mathbf{x}_1) + (1-\alpha)f(\mathbf{x}_2). \tag{A.1}$$

f is said to be *strictly convex* if the inequality in Equation (A.1) is strict for $0 < \alpha < 1$.

Geometrically, a function is convex if the line joining two points on its graph is always above the graph. Examples of convex and nonconvex functions are shown in Figure A.2.

A.3 The Convex Programming Problem

The convex programming problem is stated as follows:

$$\textit{minimize} \qquad f(\mathbf{x}) \tag{A.2}$$

$$\textit{such that} \quad \mathbf{x} \in S \tag{A.3}$$

where f is a convex function and S is a convex set.

This problem has the property that any local minimum of f over S is a global minimum, thereby easing the optimization process since it is unnecessary to perform hill-climbing out of local minima. A convex programming formulation can be solved in polynomial time [68],[69].

A.4 Posynomial Functions

A *posynomial* is a function g of a positive variable $\mathbf{x} \in \mathbf{R}^{\mathrm{n}}$ that has the form

$$g(\mathbf{x}) = \sum_j \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \tag{A.4}$$

where the exponents $\alpha_{ij} \in \mathbf{R}$ and the coefficients $\gamma_j > 0$. A posynomial is a function that is similar to a polynomial, except that

- The coefficients γ_j must be positive.

- An exponent α_{ij} could be any real number, and not necessarily a positive integer, unlike the case of polynomials.

A posynomial has the useful property that it can be mapped onto a convex function through an elementary variable transformation [83]

$$(x_i) = (e^{z_i}) \tag{A.5}$$

Such a functional form is very desirable, since such a transformation maps the problem of minimizing a posynomial function under posynomial constraints to a convex programming problem (which can be solved in polynomial time).

For example, the function $3.7x_1^{1.4}x_2^{\sqrt{3}} + 1.8x_1^{-1}x_3^{2.3}$ is a posynomial in the variables x_1, x_2, x_3. A few other examples include:

$$f(x_i) = 1/x_i \tag{A.6}$$

$$f(x_i) = constant \tag{A.7}$$

$$f(x_i, x_j) = \frac{x_i}{x_j} \tag{A.8}$$

$$f(x_i) = x_i \tag{A.9}$$

A.5 A Few Properties of Convex and Posynomial Functions

If f and g are convex functions defined on a convex set S, then the following properties hold:

Sum Property The function $f + g$ is a convex function over S.

Constant Property The function $c \cdot f$, where c is a nonnegative constant, is a convex function over S.

Max Property The function $\max(f, g)$ is a convex function over S.

Min Property The function $\min(f, g)$ is a convex function over S.

As shown before, posynomials can be transformed to convex functions using Equation (A.5). Therefore, given two posynomial functions h and j defined on S, all the above properties hold for the pair.

APPENDIX B

CODE FOR BENCHMARKS

In this appendix, we have included extended HPF and MATLAB codes for our five benchmark applications – Cmmul, Strassen, Cfd, Poly and Bicg.

```
REAL AREAL(256,256),AIMAG(256,256)
REAL BREAL(256,256),BIMAG(256,256)
REAL CREAL(256,256),CIMAG(256,256)
REAL TEMP1(256,256),TEMP2(256,256)
REAL TEMP3(256,256),TEMP4(256,256)
CHPF$   PROCESSES
CALL READMATRIX(AREAL)
CALL READMATRIX(AIMAG)
CALL READMATRIX(BREAL)
CALL READMATRIX(BIMAG)
CALL MATRIXMULTIPLY(AREAL,BREAL,TEMP1)
CALL MATRIXMULTIPLY(AIMAG,BIMAG,TEMP2)
CALL MATRIXMULTIPLY(AREAL,BIMAG,TEMP3)
CALL MATRIXMULTIPLY(AIMAG,BREAL,TEMP4)
CALL MATRIXSUBTRACT(TEMP1,TEMP2,CREAL)
CALL MATRIXADD(TEMP3,TEMP4,CIMAG)
CHPF$   ENDPROCESSES
END
```

Extended HPF Code for Cmmul

```
%! local float areal(256,256),aimag(256,256);
%! local float breal(256,256),bimag(256,256);
%! local float creal(256,256),cimag(256,256);
areal=readmatrix(areal);
aimag=readmatrix(aimag);
breal=readmatrix(breal);
bimag=readmatrix(bimag);
creal = areal*breal-aimag*bimag;
cimag = areal*bimag+aimag*breal;
```

MATLAB Code for Cmmul

```
      REAL A11(256,256),A12(256,256),A21(256,256),A22(256,256)
      REAL B11(256,256),B12(256,256),B21(256,256),B22(256,256)
      REAL TEMP1(256,256),TEMP2(256,256),TEMP3(256,256),TEMP4(256,256)
      REAL TEMP5(256,256),TEMP6(256,256),TEMP7(256,256),TEMP8(256,256)
      REAL TEMP9(256,256),TEMP10(256,256)
      REAL Q1(256,256),Q2(256,256),Q3(256,256),Q4(256,256)
      REAL Q5(256,256),Q6(256,256),Q7(256,256)
      REAL C11(256,256),C12(256,256),C21(256,256),C22(256,256)
CHPF$ PROCESSES
      CALL READMATRIX(A11)
      CALL READMATRIX(A12)
      CALL READMATRIX(A21)
      CALL READMATRIX(A22)
      CALL READMATRIX(B11)
      CALL READMATRIX(B12)
      CALL READMATRIX(B21)
      CALL READMATRIX(B22)
      CALL MATRIXADD(A11,A22,TEMP1)
      CALL MATRIXADD(B11,B22,TEMP2)
      CALL MATRIXADD(A21,A22,TEMP3)
      CALL MATRIXSUBTRACT(B12,B22,TEMP4)
      CALL MATRIXSUBTRACT(B21,B11,TEMP5)
      CALL MATRIXADD(A11,A12,TEMP6)
      CALL MATRIXSUBTRACT(A21,A11,TEMP7)
      CALL MATRIXSUBTRACT(A12,A22,TEMP8)
      CALL MATRIXADD(B11,B12,TEMP9)
      CALL MATRIXADD(B21,B22,TEMP10)
      CALL MATRIXMULTIPLY(TEMP1,TEMP2,Q1)
      CALL MATRIXMULTIPLY(TEMP3,B11,Q2)
      CALL MATRIXMULTIPLY(A11,TEMP4,Q3)
      CALL MATRIXMULTIPLY(A22,TEMP5,Q4)
      CALL MATRIXMULTIPLY(TEMP6,B22,Q5)
      CALL MATRIXMULTIPLY(TEMP7,TEMP8,Q6)
      CALL MATRIXMULTIPLY(TEMP9,TEMP10,Q7)
      CALL MATRIXADD(Q1,Q4,TEMP1)
      CALL MATRIXSUBTRACT(Q7,Q5,TEMP2)
      CALL MATRIXADD(TEMP1,TEMP2,C11)
      CALL MATRIXADD(Q2,Q4,C21)
      CALL MATRIXADD(Q3,Q5,C12)
      CALL MATRIXADD(Q1,Q3,TEMP3)
      CALL MATRIXSUBTRACT(Q6,Q2,TEMP4)
      CALL MATRIXADD(TEMP3,TEMP4,C22)
CHPF$ ENDPROCESSES
      END
```

Extended HPF Code for Strassen

```
%! local float a11(256,256),a12(256,256),a21(256,256),a22(256,256);
%! local float b11(256,256),b12(256,256),b21(256,256),b22(256,256);
%! local float Q1(256,256),Q2(256,256),Q3(256,256),Q4(256,256);
%! local float Q5(256,256),Q6(256,256),Q7(256,256);
%! local float c11(256,256),c12(256,256),c21(256,256),c22(256,256);
a11=readmatrix(a11);
a12=readmatrix(a12);
a21=readmatrix(a21);
a22=readmatrix(a22);
b11=readmatrix(b11);
b12=readmatrix(b12);
b21=readmatrix(b21);
b22=readmatrix(b22);
Q1=(a11+a22)*(b11+b22);
Q2=(a21+a22)*b11;
Q3=a11*(b12-b22);
Q4=a22*(b21-b11);
Q5=(a11+a12)*b22;
Q6=(a21-a11)*(b11+b12);
Q7=(a12-a22)*(b21-b22);
c11=Q1+Q4-Q5+Q7;
c21=Q2+Q4;
c12=Q3+Q5;
c22=Q1+Q3-Q2+Q6;
```

MATLAB Code for Strassen

```
      REAL W1I(256,256),W1R(256,256)
      REAL W2I(256,256),W2R(256,256)
      REAL W3I(256,256),W3R(256,256)
      REAL SUI(256,256),SUR(256,256)
      REAL SVI(256,256),SVR(256,256)
      REAL SWI(256,256),SWR(256,256)
      REAL DUI(256,256),DUR(256,256)
      REAL DVI(256,256),DVR(256,256)
      REAL DWI(256,256),DWR(256,256)
CHPF$ PROCESSES
      CALL READMATRIX(W1R)
      CALL READMATRIX(W1I)
      CALL READMATRIX(W2R)
      CALL READMATRIX(W2I)
      CALL READMATRIX(W3R)
      CALL READMATRIX(W3I)
      CALL READMATRIX(SUR)
      CALL READMATRIX(SUI)
      CALL READMATRIX(SVR)
      CALL READMATRIX(SVI)
      CALL READMATRIX(SWR)
      CALL READMATRIX(SWI)
      CALL FFT2D(W1R,W1I)
      CALL FFT2D(W2R,W2I)
      CALL FFT2D(W3R,W3I)
      CALL FFT2D(SUR,SUI)
      CALL FFT2D(SVR,SVI)
      CALL FFT2D(SWR,SWI)
      CALL ARRAYMULTIPLY(SVR,W3R,TEMP1)
      CALL ARRAYMULTIPLY(SWR,W2R,TEMP2)
      CALL MATRIXSUBTRACT(TEMP1,TEMP2,DUR);
      CALL ARRAYMULTIPLY(SWR,W1R,TEMP3)
      CALL ARRAYMULTIPLY(SUR,W3R,TEMP4)
      CALL MATRIXSUBTRACT(TEMP3,TEMP4,DVR);
      CALL ARRAYMULTIPLY(SUR,W2R,TEMP5)
      CALL ARRAYMULTIPLY(SVR,W1R,TEMP6)
      CALL MATRIXSUBTRACT(TEMP5,TEMP6,DWR);
      CALL FFT2D(DUR,DUI)
      CALL FFT2D(DVR,DVI)
      CALL FFT2D(DWR,DWI)
CHPF$ ENDPROCESSES
      END
```

Extended HPF Code for CFD

```
%! local float w1I(256,256),w1R(256,256);
%! local float w2I(256,256),w2R(256,256);
%! local float w3I(256,256),w3R(256,256);
%! local float suI(256,256),suR(256,256);
%! local float svI(256,256),svR(256,256);
%! local float swI(256,256),swR(256,256);
%! local float duI(256,256),duR(256,256);
%! local float dvI(256,256),dvR(256,256);
%! local float dwI(256,256),dwR(256,256);
w1R=readmatrix(w1R);
w1I=readmatrix(w1I);
w2R=readmatrix(w2R);
w2I=readmatrix(w2I);
w3R=readmatrix(w3R);
w3I=readmatrix(w3I);
suR=readmatrix(suR);
suI=readmatrix(suI);
svR=readmatrix(svR);
svI=readmatrix(svI);
swR=readmatrix(swR);
swI=readmatrix(swI);
[w1R,w1I] = fft2d(w1R,w1I);
[w2R,w2I] = fft2d(w2R,w2I);
[w3R,w3I] = fft2d(w3R,w3I);
[suR,suI] = fft2d(suR,suI);
[svR,svI] = fft2d(svR,svI);
[swR,swI] = fft2d(swR,swI);
duR = svR.*w3R-swR.*w2R;
duI = zero(duI);
dvR = swR.*w1R-suR.*w3R;
dvI = zero(dvI);
dwR = suR.*w2R-svR.*w1R;
dwI = zero(dwI);
[duR,duI] = fft2d(duR,duI);
[dvR,dvI] = fft2d(dvR,dvI);
[dwR,dwI] = fft2d(dwR,dwI);
```

MATLAB Code for CFD

```
        REAL AR(256,256),AI(256,256)
        REAL BR(256,256),BI(256,256)
        REAL CR(256,256),CI(256,256)
        INTEGER I
CHPF$   PROCESSES
        DO I=1,16
          CALL READMATRIX(AR)
          CALL READMATRIX(AI)
          CALL READMATRIX(BR)
          CALL READMATRIX(BI)
          CALL FFT2D(AR,AI)
          CALL FFT2D(BR,BI)
          CALL ARRAYMULTIPLY(AR,BR)
          CALL ARRAYMULTIPLY(AI,BI)
          CALL FFT2D(CR,CI)
        ENDDO
CHPF$   ENDPROCESSES
        END
```

Extended HPF Code for POLY

```
%! local float ar(256,256),ai(256,256);
%! local float br(256,256),bi(256,256);
%! local float cr(256,256),ci(256,256);
%! local int i;
for i=1:16
  ar=readmatrix(ar);
  ai=readmatrix(ai);
  br=readmatrix(br);
  bi=readmatrix(bi);
  [ar,ai]=fft2d(ar,ai);
  [br,bi]=fft2d(br,bi);
  cr=ar.*br;
  ci=ai.*bi;
  [cr,ci]=fft2d(cr,ci);
end
```

MATLAB Code for POLY

```
      REAL A(512,512),A1(512,512),M(512,512),M1(512,512),X(512),X1(512)
      REAL ERROR,ERROR1,BNRM2,R(512),R_TLD(512),Z(512),Z_TLD(512),B(512)
      REAL P(512),P_TLD(512),Q(512),Q_TLD(512),RHO,RHO_1,RHO_T
CHPF$ PROCESSES
      CALL READMATRIX(A)
      CALL READMATRIX(M)
      CALL READVECTOR(X)
      CALL READVECTOR(B)
      CALL NORM(B,BNRM2)
      IF (BNRM2.NE.0) THEN
        CALL MATRIXVECTORMULTIPLY(A,X,X1)
        CALL VECTORADD(B,X1,R)
        CALL NORM(R,ERROR1)
        IF ((ERROR1/BNRM2).GT.0.1)
          CALL VECTORCOPY(R_TLD,R)
          CALL EQUATIONSOLVE(M,R,Z)
          CALL EQUATIONSOLVE(M1,R_TLD,Z_TLD)
          CALL DOTPRODUCT(Z,R_TLD,RHO)
          IF (RHO.NE.0) THEN
            CALL VECTORCOPY(Z,P)
            CALL VECTORCOPY(Z_TLD,P_TLD)
            CALL MATRIXVECTORMULTIPLY(A,P,Q)
            CALL MATRIXVECTORMULTIPLY(A1,P_TLD,Q_TLD)
            CALL DOTPRODUCT(P_TLD,Q,RHO_T)
            CALL VECTORSCALE(P,(RHO/RHO_T),X1)
            CALL VECTORADD(X,X1,X)
            CALL VECTORSCALE(Q,(RHO/RHO_T),X1)
            CALL MINUS(R,X1,R)
            CALL VECTORSCALE(Q_TLD,(RHO/RHO_T),X1)
            CALL MINUS(R_TLD,X1,R_TLD)
            CALL NORM(R,ERROR)
            DO WHILE ((ERROR/BNRM2).GT.0.1)
              RHO_1 = RHO
              CALL EQUATIONSOLVE(M,R,Z)
              CALL EQUATIONSOLVE(M1,R_TLD,Z_TLD)
              CALL DOTPRODUCT(Z,R_TLD,RHO)
              IF (RHO.NE.0) THEN
                CALL VECTORSCALE(P,(RHO/RHO_1),X1)
                CALL VECTORADD(Z,X1,P)
                CALL VECTORSCALE(P_TLD,(RHO/RHO_1),X1)
                CALL VECTORADD(Z_TLD,X1,P_TLD)
                CALL MATRIXVECTORMULTIPLY(A,P,Q)
                CALL MATRIXVECTORMULTIPLY(A1,P_TLD,Q_TLD)
                CALL DOTPRODUCT(P_TLD,Q,RHO_T)
                CALL VECTORSCALE(P,(RHO/RHO_T),X1)
                CALL VECTORADD(X,X1,X)
                CALL VECTORSCALE(Q,(RHO/RHO_T),X1)
                CALL MINUS(R,X1,R)
                CALL VECTORSCALE(Q_TLD,(RHO/RHO_T),X1)
                CALL MINUS(R_TLD,X1,R_TLD)
                CALL NORM(R,ERROR)
              ENDIF
            ENDDO
          ENDIF
        ENDIF
      ENDIF
CHPF$ ENDPROCESSES
      END
```

Extended HPF Code for Bicg

```
%! local float A(512,512),M(512,512);
%! local float x(512,1),b(512,1);
%! local float 0.1,error,bnrm2;
%! local float r(512,1),r_tld(512,1),z(512,1),z_tld(512,1);
%! local float p(512,1),p_tld(512,1),q(512,1),q_tld(512,1);
%! local float alpha,beta,rho,rho_1;
A = readmatrix(A);
M = readmatrix(M);
x = readvector(x);
b = readvector(b);
bnrm2 = norm(b);
if (bnrm2 ~= 0)
  r = b - matvec(A,x);
  error1 = norm(r) / bnrm2;
  if (error1 > 0.1)
    r_tld = r;
    z = solve(M,r);
    z_tld = solve(M',r_tld);
    rho = dotp(z,r_tld);
    if (rho ~= 0)
      p = z;
      p_tld = z_tld;
      q = matvec(A,p);
      q_tld = matvec(A',p_tld);
      alpha = rho / dotp(p_tld,q);
      x = x + vecscale(p,alpha);
      r = r - vecscale(q,alpha);
      r_tld = r_tld - vecscale(q_tld,alpha);
      error = norm(r) / bnrm2;
      while (error > 0.1)
        rho_1 = rho;
        z = solve(M,r);
        z_tld = solve(M',r_tld);
        rho = dotp(z,r_tld);
        if (rho ~= 0)
          beta = rho / rho_1;
          p = z + vecscale(p,beta);
          p_tld = z_tld + vecscale(p_tld,beta);
          q = matvec(A,p);
          q_tld = matvec(A',p_tld);
          alpha = rho / dotp(p_tld,q);
          x = x + vecscale(p,alpha);
          r = r - vecscale(q,alpha);
          r_tld = r_tld - vecscale(q_tld,alpha);
          error = norm(r) / bnrm2;
        end
      end
    end
  end
end
```

MATLAB Code for Bicg

REFERENCES

[1] High Performance Fortran Forum, "High Performance Fortran Language Specification, version 1.1," Tech. Rep., Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1994.

[2] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994.

[3] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Multicomputers," *IEEE Computer*, vol. 28, no. 10, pp. 37–47, Oct. 1995.

[4] The Mathworks, Inc., Natick, MA, *MATLAB Reference Guide*, 1992.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK : A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers," in *Frontiers '92 : Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Oct. 1992, pp. 120–127.

[6] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, "LAPACK Working Note 80 : The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," Tech. Rep. UT-CS-94-246, University of Tennessee, Knoxville, TN, Sept. 1994.

[7] I. Foster, B. Avalani, A. Choudhary, and M. Xu, "A Compilation System That Integrates High Performance Fortran and Fortran M," in *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, pp. 293–300.

[8] T. Gross, D. O'Halloran, and J. Subhlok, "Task Parallelism in a High Performance Fortran Framework," *IEEE Parallel and Distributed Technology*, vol. 2, no. 3, pp. 16–26, Fall 1994.

[9] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling Fortran D for MIMD Distributed Memory Machines," *Communications of the ACM*, vol. 35, no. 8, pp. 66–80, Aug. 1992.

[10] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim, "An Overview of a Compiler for Scalable Parallel Machines," in *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993, pp. 253–272.

[11] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Y. Wang, W.-M. Ching, and T. Ngo, "An HPF compiler for the IBM SP2," in *Proceedings of Supercomputing*, San Diego, CA, Dec. 1995.

[12] Digital Equipment Corporation, Maynard, MA, *Digital High Performance Fortran 90 HPF and PSE Manual*, 1995.

[13] Portland Group Inc., Wilsonville, OR, *PGHPF User's Guide*, 1995.

[14] Applied Parallel Research, Placerville, CA, *XHPF User's Guide, Version 2.0*, 1995.

[15] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, vol. 1, no. 1, pp. 31–50, Aug. 1992.

[16] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu, "Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers," *Journal of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 15–26, Apr. 1994.

[17] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su, "The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers," in *Proceedings of the First International Workshop on Parallel Processing*, Bangalore, India, Dec. 1994, pp. 322–330.

[18] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 179–193, Mar. 1992.

[19] M. Gupta and P. Banerjee, "PARADIGM: A Compiler for Automated Data Distribution on Multicomputers," in *Proceedings of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[20] M. Gupta, "Automatic Data Partitioning on Distributed Memory Multicomputers, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, Sept. 1992, CRHC-92-19/UILU-ENG-92-2237.

[21] D. J. Palermo and P. Banerjee, "Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers," in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995.

[22] E. Su, D. J. Palermo, and P. Banerjee, "Automating Parallelization of Regular Computations for Distributed Memory Multicomputers in the PARADIGM Compiler," in *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1993, pp. II:30–38.

[23] E. Su, "Automating Parallelization of Regular Computations for Distributed-Memory Multicomputers," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL, 1993.

[24] D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, "Compiler Optimizations for Distributed Memory Multicomputers Used in the PARADIGM Compiler," in *Proceedings of the 23rd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994, pp. II:1–10.

[25] E. Su, D. J. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed-Memory Multicomputers," in *Proceedings of the 1994 International Conference on Parallel Architectures and Compilation Techniques*, Montréal, Canada, Aug. 1994, pp. 123–132.

[26] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges IV, and P. Banerjee, "Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers," in *Proceedings of the 9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995, pp. 424–433.

[27] E. W. Hodges IV, "High Performance Fortran Support for the PARADIGM Compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, Oct. 1995, CRHC-95-23/UILU-ENG-95-2237.

[28] A. Lain and P. Banerjee, "Techniques to Overlap Computation and Communication in Irregular Iterative Applications," in *Proceedings of the 8th ACM International Conference on Supercomputing*, Manchester, England, July 1994, pp. 236–245.

[29] A. Lain and P. Banerjee, "Exploiting Spatial Regularity in Irregular Iterative Applications," in *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995, pp. 820–827.

[30] A. Lain, "Compiler and Run-Time Support for Irregular Computations, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, IL, Oct. 1995, CRHC-92-22.

[31] K. P. Belkhale and P. Banerjee, "Approximate Algorithms for the Partitionable Independent Task Scheduling Problem," in *Proceedings of the 19th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1990, pp. 72–75.

[32] K. P. Belkhale and P. Banerjee, "A Scheduling Algorithm for Parallelizable Dependent Tasks," in *Proceedings of the International Parallel Processing Symposium*, Anaheim, CA, Apr. 1991, pp. 500–506.

[33] S. Ramaswamy and P. Banerjee, "Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers," in *Proceedings of the 22nd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1993, pp. II:134–138.

[34] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," in *Proceedings of the 23rd International Conference on Parallel Processing*, St. Charles, IL, Aug. 1994, pp. II:116–125.

[35] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," Tech. Rep. CRHC-94-10/UILU-ENG-94-2224, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, June 1994.

[36] S. Ramaswamy and P. Banerjee, "Simultaneous Allocation and Scheduling Using Convex Programming Techniques," *Parallel Processing Letters*, Dec. 1995, to appear.

[37] E. Anderson, Z. Bai, C. Bischof, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorenson, "LAPACK : A Portable Linear Algebra Library for High-

performance Computers," in *Proceedings of Supercomputing*, New York, NY, Nov. 1990, pp. 2–11.

[38] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines - EISPACK Guide*, vol. 6 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Germany, 1976.

[39] J. J. Dongarra, J. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979.

[40] J. J. Dongarra and R. C. Whaley, "LAPACK Working Note 94: A User's Guide to the BLACS v1.0," Tech. Rep. UT-CS-94-281, University of Tennessee, Knoxville, TN, Mar. 1995.

[41] J. Subhlok, J. Stichnoth, D. O'Halloran, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993, pp. 13–22.

[42] P. Dinda, T. Gross, D. O'Halloran, E.Segall, J. Stichnoth, J. Subhlok, J. Webb, and B. Yang, "The CMU Task Parallel Program Suite," Tech. Rep. CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Mar. 1994.

[43] I. Foster and K. M. Chandy, "Fortran M: A Language for Modular Parallel Programming," *Journal of Parallel and Distributed Computing*, vol. 26, no. 1, pp. 24–35, Apr. 1995.

[44] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," in *Proceedings of the 18th International Conference on Parallel Processing*, St. Charles, IL, Aug. 1989, pp. 39–48.

[45] M. Girkar and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 166–178, Mar. 1992.

[46] M. Girkar, "Functional Parallelism : Theoretical Foundations and Implementations, Ph.D. thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Dec. 1991, CSRD-1182.

[47] J. E. Moreira and C. D. Polychronopoulos, "On the Implementation and Effectiveness of Autoscheduling," Tech. Rep. CSRD-1372, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1994.

[48] J. E. Moreira and C. D. Polychronopoulos, "Autoscheduling in a Distributed Shared-Memory Environment," Tech. Rep. CSRD-1373, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, May 1994.

[49] J. E. Moreira, "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors, Ph.D. thesis, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Jan. 1995.

[50] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua, "A MATLAB Compiler and Restructurer for the Development of Scientific Libraries and Applications," in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, Aug. 1995, pp. 18.1–18.18.

[51] L. De Rose and D. Padua, "Inference Mechanism for Fast Array Language Computation," Tech. Rep. CSRD-1462, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Dec. 1995.

[52] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, vol. 26, no. 1, pp. 22–35, Jan. 1978.

[53] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979.

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

[55] T. Yang and A. Gerasoulis, "A Fast Statis Scheduling Algorithm for DAGs on an Unbounded Number of Processors," in *Proceedings of Supercomputing*, Albuquerque, NM, Nov. 1991, pp. 633–642.

[56] T. Yang and A. Gerasoulis, "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors," in *Proceedings of the Scalable High Performance Computing Conference*, Williamsburg, VA, Apr. 1992, pp. 350–357.

[57] G. N. S. Prasanna and A. Agarwal, "Compile-time Techniques for Processor Allocation in Macro Dataflow Graphs for Multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, Aug. 1992, pp. 279–283.

[58] G. N. S. Prasanna, A. Agarwal, and B. R. Musicus, "Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 7, pp. 720–736, July 1994.

[59] J. Subhlok, D. O'Halloran, T. Gross, P. Dinda, and J. Webb, "Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs," in *Proceedings of Supercomputing '94*, Washington D.C., Nov. 1994, pp. 330–339.

[60] J. Subhlok and G. Vondran, "Optimal Mapping of Sequences of Data Parallel Tasks," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995, pp. 134–143.

[61] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993, pp. 126–138.

[62] G. Agrawal, A. Sussman, and J. Saltz, "Compiler and Runtime Support for Structured and Block Structured Applications," in *Proceedings of Supercomputing*, Portland, OR, Nov. 1993, pp. 578–587.

[63] R. Thakur, A. Choudhary, and G. Fox, "Redistribution of Arrays in HPF," in *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994, pp. 309–318.

[64] F. Coelho and C. Ancourt, "Optimal compilation of hpf remappings," Tech. Rep. A-277-CRI, Centre de Recherche en Informatique, Fontainebleau, France, Oct. 1995.

[65] R. Barett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods*, SIAM, 1994.

[66] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten, "Parafrase-2 Manual," Tech. Rep., Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, Aug. 1990.

[67] M. Gupta and P. Banerjee, "Compile-Time Estimation of Communication Costs on Multicomputers," in *Proceedings of 6th International Parallel Processing Symposium*, Beverley Hills, CA, Mar. 1992, pp. 470–475.

[68] P. M. Vaidya, "A New Algorithm for Minimizing Convex Functions Over Convex Sets," in *Proceedings of the Symposium on Foundations of Computer Science*, Research Triangle Park, NC, Oct. 1989, pp. 332–337.

[69] S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. M. Kang, "An Exact Solution to the Transistor Sizing Problem for CMOS Circuits Using Convex Optimization," *IEEE Transactions on Computer Aided Design*, vol. 12, no. 11, pp. 1621–1634, Nov. 1993.

[70] C. L. Liu, *Elements of Discrete Mathematics*, McGraw-Hill, New York, NY, 1985.

[71] M. R. Garey, R. L. Graham, and D. S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Operations Research*, vol. 26, no. 1, pp. 3–21, Jan. 1978.

[72] Q. Wang and K. H. Cheng, "A Heuristic for Scheduling Parallel Tasks and Its Analysis," *SIAM Journal on Computing*, vol. 21, no. 2, pp. 281–294, Apr. 1992.

[73] Message-Passing Interface Forum, "MPI: A Message-Passing Interface Standard," Tech. Rep. CS-94-230, University of Tennessee, Knoxville, TN, 1994.

[74] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, MA, 1994.

[75] S. Hiranandani, K. Kennedy, and C-W. Tseng, "Compiler Optimizations for Fortran D on MIMD Distributed-memory Machines," in *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991, pp. 86–100.

[76] C. W. Tseng, "An optimizing fortran D compiler for MIMD distributed-memory machines, Ph.D. thesis, Rice University, Houston, TX, Jan. 1993, COMP TR93-199.

[77] Utpal Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*, Kluwer Academic Publishers, Boston, MA, 1993.

[78] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S. H. Teng, "Generating Local Addresses and Communication Sets for Data-Parallel Programs," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, San Diego, CA, May 1993, pp. 149–158.

[79] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C : The Art of Scientific Computing*, Cambridge University Press, Cambridge, England, 1988.

[80] S. L. Lyons, T. J. Hanratty, and J. B. McLaughlin, "Large-scale Computer Simulation of Fully Developed Channel Flow with Heat Transfer," *International Journal of Numerical Methods for Fluids*, vol. 13, no. 8, pp. 999–1028, Nov. 1991.

[81] M. Dhagat, R. Bagrodia, and M. Chandy, "Integrating Task and Data Parallelism in UC," in *Proceedings of the International Conference on Parallel Processing*, Oconomowoc, WI, Aug. 1995, pp. 29–36.

[82] D. G. Luenberger, *Linear and Nonlinear Programming*, Addison-Wesley, Reading, MA, 1984.

[83] J. G. Ecker, "Geometric Programming : Methods, Computations and Applications," *SIAM Review*, vol. 22, no. 3, pp. 338–362, July 1980.

VITA

Shankar Ramaswamy received the B.S. degree in Electronics from the University of Delhi in 1987. He received the M.E. degree in Electrical Engineering from the Indian Institute of Science in 1991. He is currently a candidate for the Ph.D. degree in Electrical Engineering at the University of Illinois at Urbana-Champaign.

Mr. Ramaswamy received the Prakashwati Memorial Award from Delhi University and the Alfred Hay Gold Medal from the Indian Institute of Science. He was the recipient of an IBM graduate fellowship while at the University of Illinois.

After completing his doctoral dissertation, Mr. Ramaswamy will become a Member of the Technical Staff at Transarc Corporation in Pittsburgh, PA. His research interests include architectures and systems software for high-performance/parallel computing.