A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers

Shankar Ramaswamy, Sachin Sapatnekar, *Member*, *IEEE,*
and Prithviraj Banerjee, *Fellow*, *IEEE*

Abstract—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 paper, 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 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.

Index Terms—Task parallel, data parallel, allocation, scheduling, HPF, distributed memory, convex programming.

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 paper, 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 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.

A typical example of the type of scientific program we are targeting is shown in Fig. 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. Fig. 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.

- *S. Ramaswamy is with Transarc Corp., 707 Grant St., Pittsburgh, PA 15219. E-mail: shankar@transarc.com.*
- *S. Sapatnekar is with the Department of Electrical Engineering, 200 Union St. SE, Minneapolis, MN 55455. E-mail: sachin@ee.umn.edu.*
- *P. Banerjee is with the Center for Parallel and Distributed Computing, Northwestern University, Room 4386 Technological Institute, 2145 Sheridan Rd., Evanston, IL 60208-3118. E-mail: banerjee@ece.nwu.edu.*

Manuscript received 11 July 1994; revised 18 Apr. 1996.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100892.

```
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
```

Fig. 1. Multiplication of complex matrices.

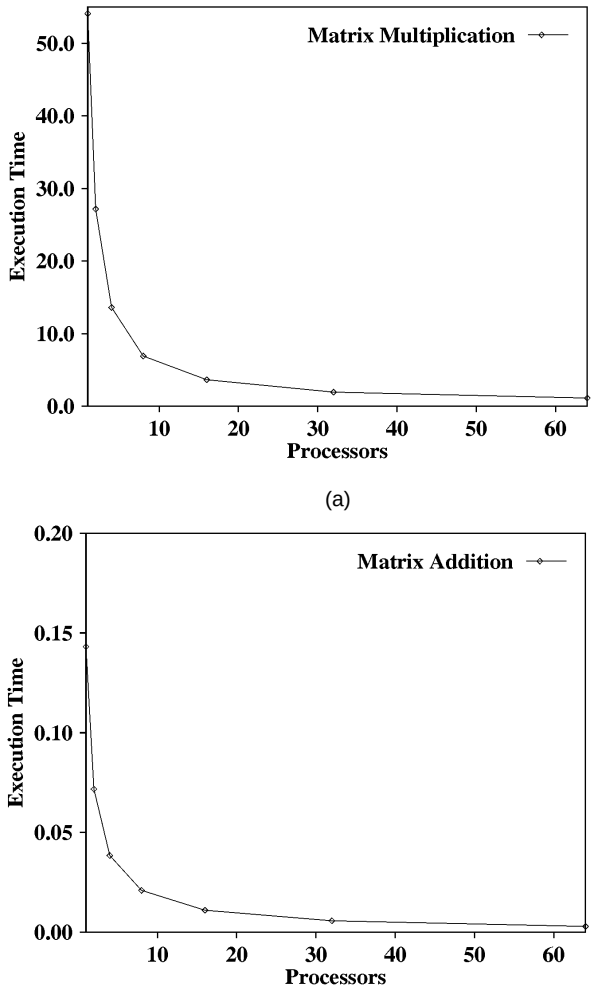


Fig. 2. Execution time for matrix multiplication and addition on the CM-5: (a) matrix multiplication, (b) matrix addition.

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. Fig. 3a illustrates this scheme of execution. The pure data parallel execution scheme finishes executing the program in 8.5 seconds.

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

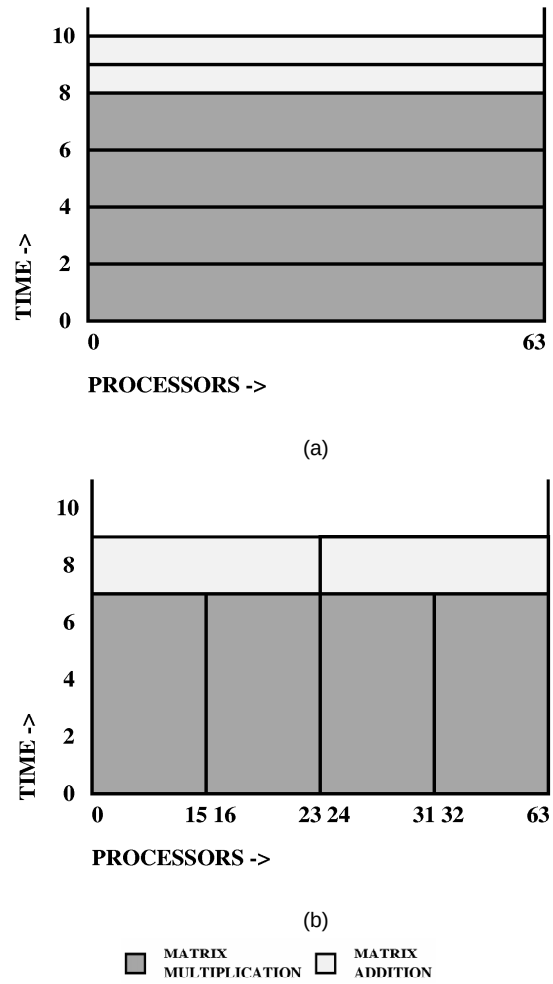


Fig. 3. Execution schemes for complex matrix multiplication program: (a) pure data parallel, (b) task and data parallel.

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 paper.

A third possible scheme of execution for the program is one that uses task and data parallelism together. Fig. 3b 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 Fig. 2 clearly shows this. When we increase the number of processors used for a multiply from 16 to 64 (a factor of four), 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 Contributions

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

- Our research has been carried out 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.
- The issues involved in automatically exploiting task and data parallelism in extended HPF applications are
  1) Detecting and representing available task and data parallelism for a given program. 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 program.
  2) Using the MDG to decide on a scheme of execution that minimizes the execution time of the given program. 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.
  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. The algorithms used for data redistribution are not discussed in this paper; details can be found in [4], [5].
- 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]. 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 [6], [7]; we have adopted some of these extensions for our HPF compiler. Section 3 discusses the 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 [8], the SUIF compiler from Stanford [9], the pHPF compiler from International Business Machines [10], the High Performance Fortran 90 compiler from Digital Equipment Corporation [11], the pgHPF compiler from the Portland Group Inc. [12], the xHPF compiler from Applied Parallel Research [13], the SUPERB compiler from the University of Vienna [14], and the FORTRAN-90D / HPF compiler from Syracuse University [15].

Fig. 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 paper 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 Section 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 Outline

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

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. We briefly discuss below a few of these research efforts.

The Fx compiler project [7], [16] targets a class of applications [17] 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 [18] was initially conceived as an extension to Fortran for expressing task parallelism. Recently, it has been integrated with HPF [6] in order to enable the simultaneous

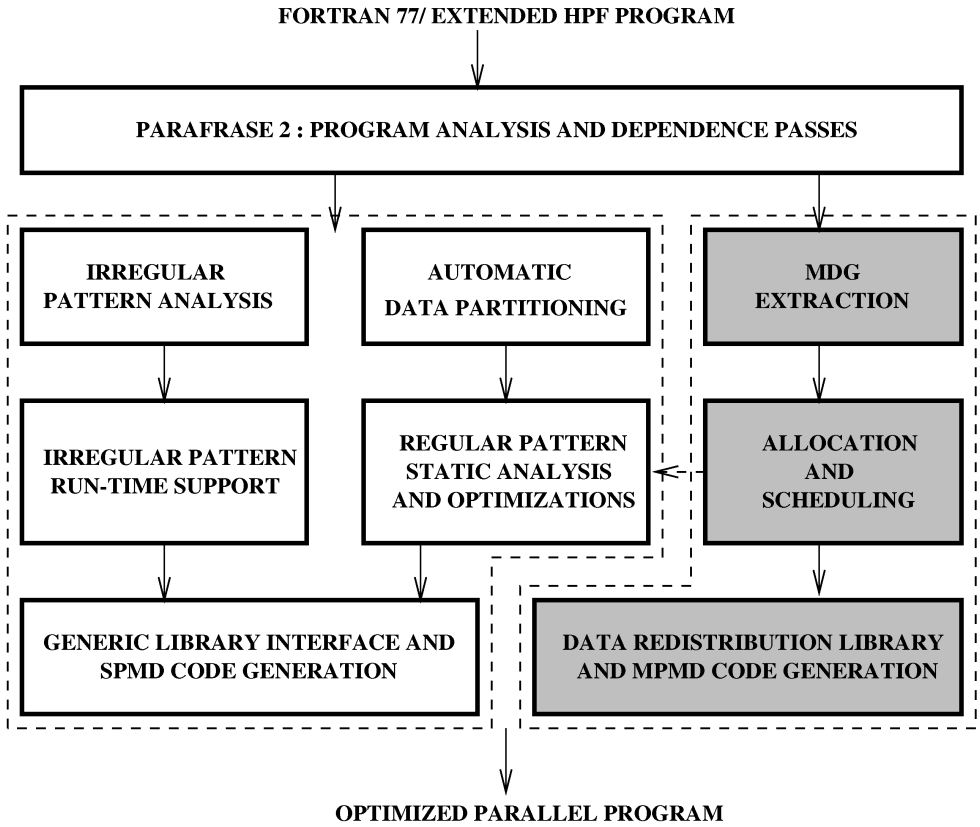


Fig. 4. Organization of the PARADIGM compiler.

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 [19] parallelizes Fortran applications for shared memory machines. It analyzes an input Fortran program and constructs a Hierarchical Task Graph (HTG) representation [20], [21] for the program. The HTG representation captures parallelism information at all levels of granularity. Based on the HTG representation, an *Autoscheduling* [22] technique has been proposed for exploiting task and data parallelism dynamically during program execution.

Dhagat et al. [23] have proposed a language, UC, that allows for the expression of task and data parallelism. The UC compiler generates code that executes the program in a task and data parallel manner. No algorithms for automatic allocation and scheduling have been implemented in the UC compiler.

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. [6] 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.
- Gross et al. [7] have proposed the the `PARALLEL` directive. The body of a `PARALLEL` section can contain HPF data parallel subroutine calls and simple `DO` loops.

For our research, we have essentially adopted a modified version of the directives proposed by Gross et al. 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 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 [20], [21]. 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.

2.4 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 [24]. Further treatment on this topic can also be found in the book by Garey and Johnson [25]. The problem of allocation and scheduling we are considering in this paper 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 [26] and by Yang and Gerasoulis [27], [28]. 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 [26]-[28] 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 [29], Prasanna et al. [30], Belkhale and Banerjee [31], [32], Subhlok et al. [33], and Subhlok and Vondran [34]. Top-down approaches take a more global view of the problem than bottom-up approaches.

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 [29], [30] 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 [33], [34] 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 [17]. It is not clear how the proposed allocation and scheduling heuristics would work for more general, nontree MDGs. Our methods are able to handle all MDGs in a uniform manner.

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 [20], [21]. Our allocation and scheduling algorithms discussed in Section 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.
- 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 Fig. 5. Fig. 5a illustrates an MDG with only `S` nodes; we refer to such MDGs as *Simple* MDGs. Fig. 5b 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 extended HPF 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

Section 3.2.1 considers the extensions required for HPF in order to facilitate the extraction of an MDG structure. Later, we present algorithms that perform the automatic extraction of the MDG structure in Section 3.2.2. 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 demarcate an MDG structure in the HPF framework, we provide the `PROCESSES` directive:

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

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

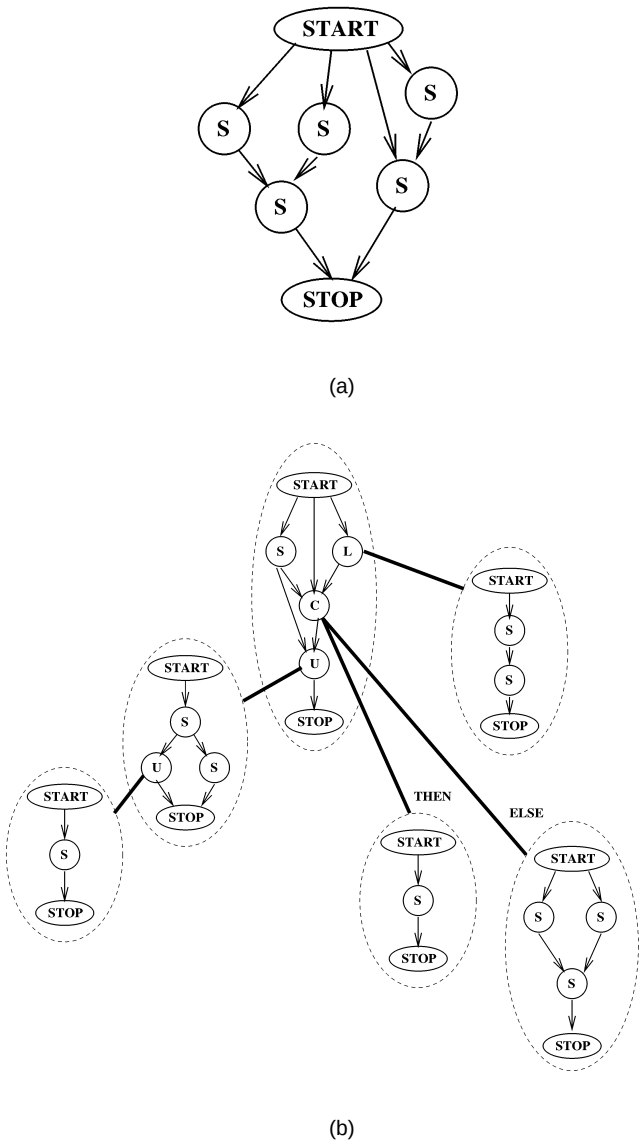


Fig. 5. Examples of MDGs: (a) simple MDG, (b) hierarchical MDG.

subroutine must be enclosed inside the PROCESSES directive. 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
```

To ease our dependence analysis algorithms, we provide another set of directives for the user. These 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:

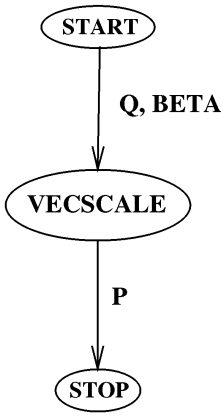
```
         SUBROUTINE MATRIXADD(A, B, C)
CHPF$    IN A,B
CHPF$    OUT C
         SUBROUTINE DOTPRODUCT(A, DP)
CHPF$    IN A
CHPF$    OUT DP
         SUBROUTINE FFT2D(AR, AR)
CHPF$    INOUT AR, AI
```

3.2.2 Extraction of MDG Structure

To extract the MDG structure for a given extended HPF application, we use Parafrase-2 [19], [35] (with modifications to accommodate extended HPF [36]) as a front end to build a parse tree representation. We then use this parse tree to construct the hierarchical MDG structure for the program. Details of algorithms used for building the MDG for various types of statements are provided in [5]; we present some simple examples here to illustrate the process.

As a first step, we traverse the parse tree and build the MDG structure for each program unit (HPF subroutine) that contains a PROCESSES directive. This is done in an incremental manner by traversing the statements inside the body of the PROCESSES directive. A separate MDG is constructed for each statement before combining it with the MDG corresponding to all previous statements.

Fig. 6 illustrates the process of building the MDG structure for a subroutine call. To build an MDG for this statement, we create an MDG with a single S node corresponding to the VECSCALE subroutine. The input variables Q and BETA cause the addition of the edge between the START




```
CALL VECSCALE (Q, BETA, P)
```

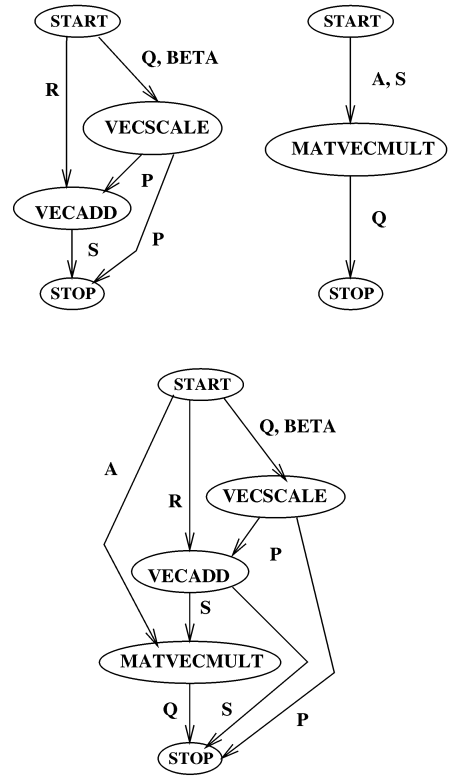
Fig. 6. Example of MDG construction for a subroutine call.

node of the MDG and node corresponding to the VECSCALE call. Similarly, the output variable P causes the addition of the edge between the VECSCALE call and the STOP node. Intuitively, we summarize the variable read and write information using edges to START and STOP. This allows us to easily combine MDGs during the incremental MDG building process for a program.

The process of combining MDGs is illustrated in Fig. 7. In order to combine MDGs, 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 can be easily computed using 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 of the first MDG. Last, we free the STOP node of the first MDG and the START node of the second MDG.

Fig. 8 illustrates the procedure used for building the hierarchical MDG structure for a conditional statement. 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 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.

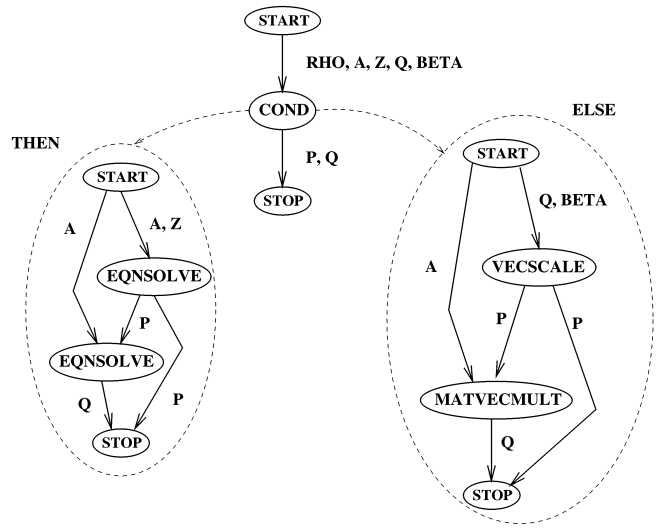
A procedure similar to the one described above is used for building the hierarchical MDG for loop statements; details of this procedure can be found in [5].




```
CALL VECSCALE (Q, BETA, P)
CALL VECADD (P, R, S)

CALL MATVECMULT (A, S, Q)
```

Fig. 7. Example of MDG combining.




```
IF (RHO .NE. O) THEN
  CALL EQNSOLVE (A, Z, P)
  CALL EQNSOLVE (A, P, Q)
ELSE
  CALL VECSCALE (Q, BETA, P)
  CALL METVECMULT (A, P, Q)
ENDIF
```

Fig. 8. Example of MDG construction for a conditional statement.

3.3 Mathematical Cost Models

In this section, we describe our processing and data transfer cost functions and prove that they all belong to the class of posynomial functions [37], [5]. An important property of posynomial functions is that they can be mapped onto the class of convex functions[1]; our allocation and scheduling algorithms described in Section 4 rely on this property.

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 $\left(t_i^C\right)$ as a function of the number of processors it uses (p_i) is given by

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

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.

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

PROOF. Due to a lack of space, we are unable to provide a proof here, please refer [5] for a complete proof. □

LEMMA 2. *$t_i^C \cdot p_i$ is a posynomial function w.r.t. p_i.*

PROOF. Due to a lack of space, we are unable to provide a proof here, please refer [5] for a complete proof. □

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 1,000 bytes each, and every receiver may receive five messages of 1,500 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.

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_{ij}^S incurred by processors allocated to node i, a network delay component t_{ij}^D to allow for messages to reach the processors allocated to node j, and a receiving component t_{ij}^R incurred by processors allocated to node j. The t_{ij}^S component is accounted for in the weight of node i, the t_{ij}^D component is taken to be the weight of the edge joining node i and node j, and the t_{ij}^R component is accounted for in the weight of node j.

We propose the following expressions for the three cost components

$$\begin{aligned} t_{ij}^S &= S_{ij}\left(p_i, p_j\right) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\ t_{ij}^D &= \frac{L}{p_i \cdot S_{ij}\left(p_i, p_j\right)} \cdot t_n \\ t_{ij}^R &= R_{ij}\left(p_i, p_j\right) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr}, \end{aligned} \tag{2}$$

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, and
- R_{ij} is the number of messages received at each receiving processor.

Intuitively, the sending component $\left(t_{ij}^S\right)$ 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 $\left(\frac{L}{p_i}\right)$. The same logic holds for the receiving component $\left(t_{ij}^R\right)$

1. Convex functions have a unique global minimum value over convex sets, and such minimization problems can be solved in polynomial time [38].

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 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 2. The quantities S_{ij} and R_{ij} can be expressed in terms of block factors and skip factors using

$$S_{ij} = \max\left(1, \frac{SF_j}{SF_i}, \frac{BF_i}{BF_j}, \frac{SF_j}{SF_i} \cdot \frac{BF_i}{BF_j}\right)$$
$$R_{ij} = \max\left(1, \frac{SF_i}{SF_j}, \frac{BF_j}{BF_i}, \frac{SF_i}{SF_j} \cdot \frac{BF_j}{BF_i}\right), \quad (3)$$

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.

TABLE 1
BLOCK FACTORS FOR VARIOUS REGULAR DISTRIBUTIONS

DISTRIBUTION	BLOCK FACTOR
ALL	L
BLOCK	$\frac{L}{p_i}$
CYCLIC	1
BLOCKCYCLIC(X)	X

TABLE 2
SKIP FACTORS FOR VARIOUS REGULAR DISTRIBUTIONS

DISTRIBUTION	SKIP FACTOR
ALL	L
BLOCK	L
CYCLIC	p_i
BLOCKCYCLIC(X)	$X \cdot p_i$

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. *t_{ij}^S, t_{ij}^R, and t_{ij}^D are posynomial functions w.r.t. p_i and p_j for all possible cases of redistributions.*

PROOF. Due to a lack of space, we are unable to provide a proof here. Please refer to [5] for a complete proof. □

LEMMA 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. Due to a lack of space, we are unable to provide a proof here. Please refer to [5] for a complete proof. □

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

$$T_i = \sum_{p \in PRED_i} t_{pi}^R + t_i^C + \sum_{s \in SUCC_i} t_{is}^S, \quad (4)$$

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

$$E_{ij} = t_{ij}^N . \quad (5)$$

Using the sum property of posynomials described in [5] we can show the node and edge costs to be posynomials.

3.3.4 MDG Cost Properties

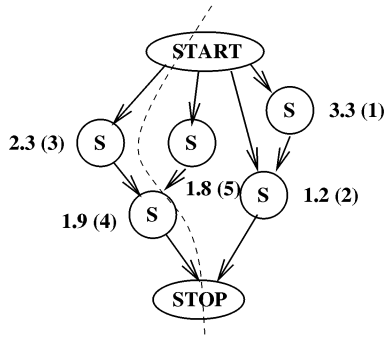
We use a few cost-related properties of an MDG in constructing our allocation and scheduling algorithms—the *Critical Path* and the *Average Area*. The critical path (C) of an MDG is defined as the time at which node n (STOP node) finishes execution. If y_i is the finish time of node i, the critical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

$$C = y_n$$
$$y_i = \max_{p \in PRED_i} \left(y_p + E_{pi}\right) + T_i . \quad (6)$$

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

$$A = \frac{1}{P} \cdot \sum_{1}^{n} T_i \cdot p_i . \quad (7)$$

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. Fig. 9 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.



CRITICAL PATH = 2.3 + 1.9 = 4.2

AVERAGE AREA = (2.3x3 + 1.9x4 + 1.8x5 + 3.3x1 + 1.2x2) / 8 = 3.65

Fig. 9. Example illustrating computation of critical path and average area.

Using the sum and max properties of posynomials [37], [5], we can show the critical path and average area to be posynomials. As we shall see in Section 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 HPF program. `S` nodes in a HPF program correspond to intrinsic calls, operators, or calls to user-defined HPF data parallel routines. For intrinsic calls and operators, we use a profiling approach for estimating the cost parameters α and τ. In the case of user-defined HPF data parallel routines, cost parameters α and τ are computed using the cost estimation techniques of Gupta [39], and Gupta and Banerjee [40]. 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.

We compute the cost model parameters for `C`, `L`, and `U` nodes by traversing the MDG in a bottom-up manner. At the lowest level in the MDG, all nodes are `S` nodes whose cost parameters we obtain as described before. Therefore, as we travel upwards in the MDG, we use cost information from lower levels to compute cost information at the upper levels. Due to a lack of space, we are unable to present the algorithms used; they are discussed in [5].

3.4 Summary

In this section, we considered the MDG representation for an application. Extensions to HPF to enable extraction of the MDG structure were discussed in Section 3.2.1. In Section 3.2.2, we considered algorithms to extract the structure of the MDG from an extended HPF 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 section, to decide on an execution scheme for the application.

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 (Section 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 4.5, we consider the allocation and scheduling algorithm for hierarchical MDGs.

4.2 Simple MDGs

For simple MDGs, we use the Two Step Allocation and Scheduling (TSAS) algorithm described in Section 4.3. 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 4.4.

4.3 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.

4.3.1 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

$$\text{minimize } \Phi = \max(A, C) \tag{8}$$

$$subject\ to\ 1 \le p_i \le P\ \forall i = 1, n\ , \tag{9}$$

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 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. Fig. 10 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 Fig. 11.

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 [37], [5]. This guarantees finding the optimum minimum solution.
2) A convex programming formulation has been shown to be solvable in polynomial time [38]—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

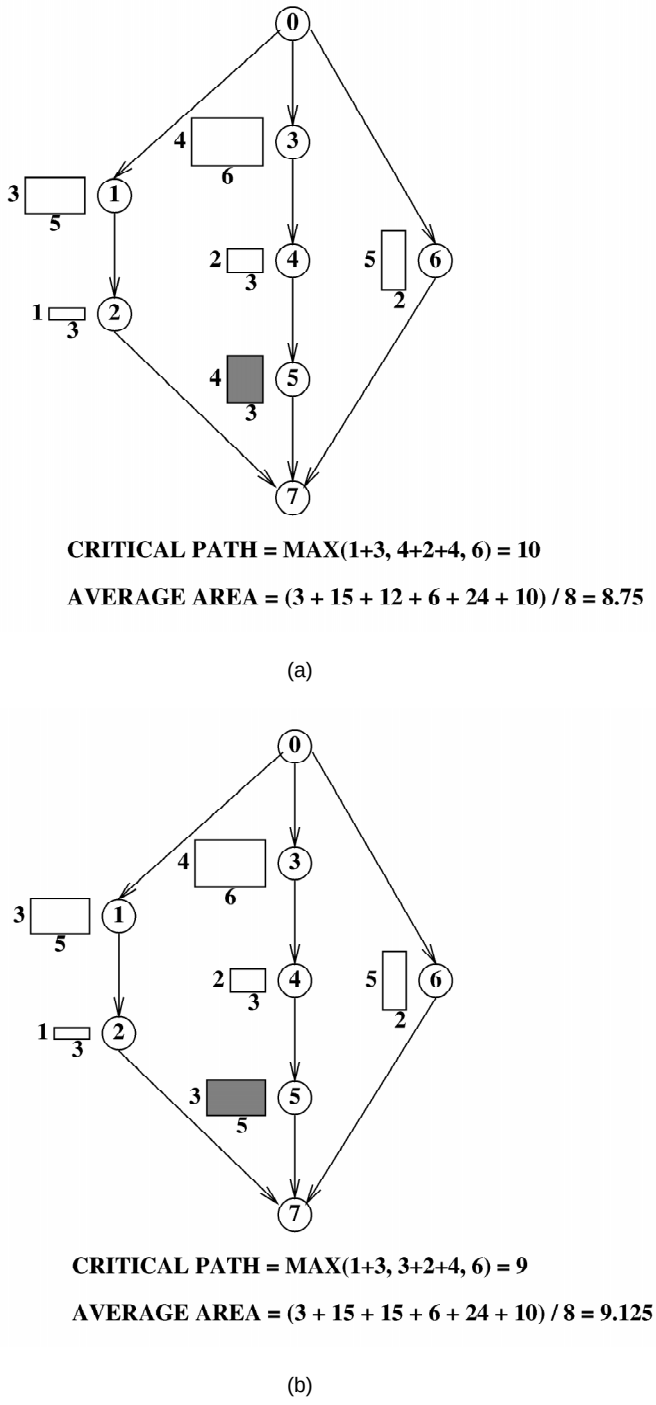


Fig. 10. Effect of allocation on critical path and average area: (a) dominating critical path, (b) dominating average area.

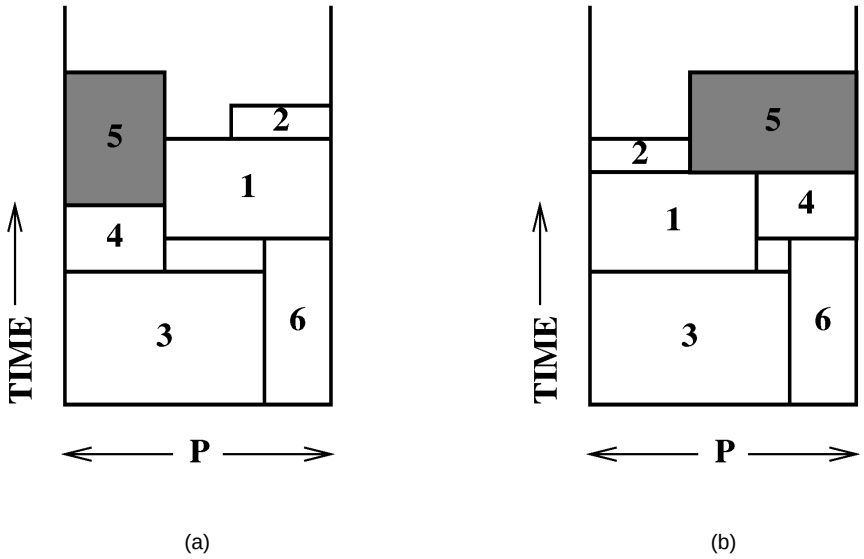


Fig. 11. Processor system bin visualization of program execution: (a) dominating critical path, (b) dominating average area.

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 4.4.

4.3.2 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 [41], Garey et al. [42], and Wang and Cheng [43]. 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 1).

The complexity of the CPAA step has been shown to be $O(n^{2.5})$ [38] 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.

4.4 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 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*.

- Theorem 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.
- Theorem 3 summarizes the effects discussed in Theorems 1 and 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 1 selects a value for *PB* that minimizes it.

THEOREM 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_{opt}^{PB} denote the value obtained using the best possible scheduler. The relationship between these two quantities is given by*

$$T_{psa} \leq \left(1 + \frac{P}{p - PB + 1}\right) \cdot T_{opt}^{PB}. \tag{10}$$

PROOF. Omitted due to lack of space. For details please refer to [44], [5]. □

THEOREM 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*

$$T_{opt}^{PB} \leq \left(\frac{2 \cdot P}{PB}\right) \cdot \Phi, \tag{11}$$

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. Omitted due to lack of space. For details please refer to [44], [5]. □

THEOREM 3. *Let T_{TSAS} denote the value of the finish time obtained for allocation and scheduling using the TSAS algorithm (Section 4.3). The deviation of T_{TSAS} from Φ is given by*

$$T_{TSAS} \leq \left(1 + \frac{P}{P - PB + 1}\right) \cdot \left(\frac{2 \cdot P}{PB}\right) \Phi, \tag{12}$$

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. Omitted due to lack of space. For details please refer to [44], [5]. □

COROLLARY 1. *The optimum value of PB to use for the TSAS algorithm given P processors is*:

$$PB = 1 + 0.58579 \cdot P. \tag{13}$$

Further, using this value of PB gives us the following worst case deviation for T_{TSAS} w.r.t. Φ:

$$T_{TSAS} \leq 11.66 \cdot \Phi. \tag{14}$$

PROOF. From Theorem 3, we see that the optimum value for *PB* is one that minimizes the following expression:

$$\text{minimize} \left(1 + \frac{P}{P - PB + 1}\right) \cdot \left(\frac{2 \cdot P}{PB}\right) \tag{15}$$

$$\textit{subject to } 1 \leq PB \leq P. \tag{16}$$

This minimization problem can easily be solved analytically for *PB* and gives us the value shown in (13). Using this value for *PB* in the bound expression shown in (15), we obtain:

$$T_{TSAS} \leq \frac{6.8284}{\frac{1}{P} + 0.58579} \cdot \Phi. \tag{17}$$

Since $P \geq 0$, we obtain:

$$T_{TSAS} \leq \frac{6.8284}{0.58579} \cdot \Phi \Rightarrow T_{TSAS} \leq 11.66 \cdot \Phi, \tag{18}$$

which is the required result. □

4.5 Hierarchical MDGs

The Two Step Allocation and Scheduling (TSAS) algorithm described in Section 4.3 forms the basis of our hierarchical allocation and scheduling algorithm. Basically, we use the TSAS algorithm to allocate and schedule at each level in a hierarchical MDG. Fig. 12 shows the Hierarchical Two Step Allocation and Scheduling (HTSAS) algorithm. The HTSAS algorithm first uses the TSAS 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 HTSAS 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 Fig. 13 to allocate and schedule the loop body. This algorithm uses unrolling to try to increase available task parallelism to improve the performance. Fig. 14 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 Fig. 13 to limit the amount of unrolling. Currently, we 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 percent of its original value.

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

```
HTSAS(MDG, NumProcs)
 1 FinishTime ← TSAS(MDG, NumProcs) MDGNodeList ← nodes[MDG]
 2 while MDGNodeList ≠ NIL
 3       do MDGNode ← node[MDGNodeList]
 4          switch
 5             case type[MDGNode] = LoopNode:
 6                  if looptype[MDGNode] = While
 7                    then HTSAS(loopmdg[MDGNode],
                                allocation[MDGNode])
 8                    else HTSASForLoop(loopmdg[MDGNode],
                                allocation[MDGNode])
 9             case type[MDGNode] = Conditional Node:
10                  HTSAS(ifmdg[MDGNode], allocation[MDGNode])
11                  HTSAS(elsemdg[MDGNode], allocation[MDGNode])
12             case type[MDGNode] = UserDefNode:
13                  HTSAS(mdg[MDGNode], allocation[MDGNode])
14                  MDGNodeList ← next[MDGNodeList]
15
16 return FinishTime
```

Fig. 12. Allocation and scheduling for hierarchical MDGs.

U **nodes**: In the case of user-defined nodes, we use the HTSAS 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.

4.6 Summary

In this section, 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 4.5.

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.

5 Implementation and Results

5.1 Introduction

In this section, we discuss experimental evidence that demonstrates the effectiveness of our proposed optimization—the simultaneous exploitation of task and data parallelism. In the next section, we briefly describe each of the benchmark applications used for our experiments. Later, we present speedup measurements obtained for our benchmark applications on the Thinking Machines CM-5 and Intel Paragon.

5.2 Benchmarks

We have used a set of five benchmark applications for our experiments. The applications are:

```
HTSASForLoop(MDG, NumProcs)
 1 FinishTime ← TSAS(MDG)
 2 OldFinishTime ← ∞
 3 UnrollFactor ← 0
 4 while (UnrollFactor < MaxFactor) and (FinishTime ≤
          ImpFactor × OldFinishTime)
 5       do OldFinishTime ← FinishTime
 6          UnrollFactor ← UnrollFactor + 1
 7          TempMDG ← UnrollMDG(MDG, UnrollFactor)
 8          FinishTime ← TSAS(MDG) / UnrollFactor
 9 Parent ← parent[MDG]
10 loopmdg[Parent] ← TempMDG
11 unrollfactor[Parent] ← UnrollFactor − 1
12 return OldFinishTime
```

Fig. 13. Allocation and scheduling for loop (L) nodes.

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 [45]. 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 [46]. 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 [23]. 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 [47]. The MDG for this application is hierarchical and contains four C nodes and an L node.

5.3 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. Fig. 15 shows speedup data obtained for the Thinking Machines CM-5. Fig. 16 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

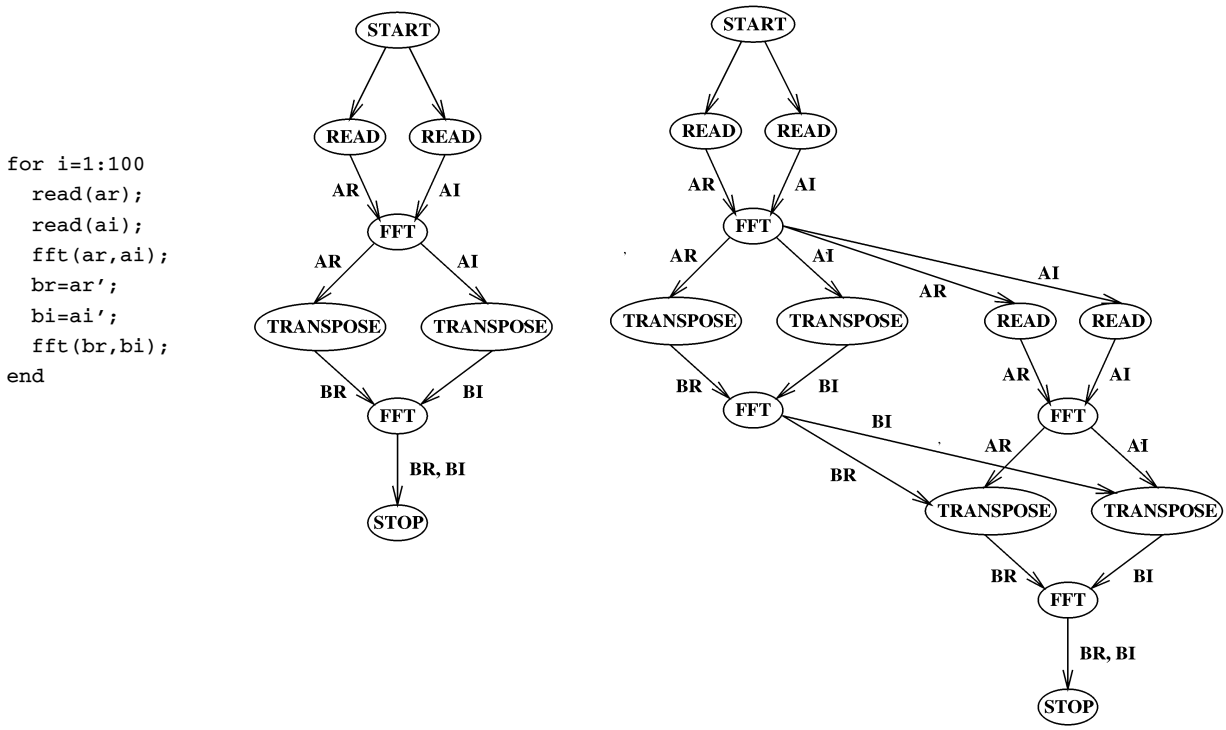


Fig. 14. Example illustrating unrolling for loops.

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 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.

6 Conclusions and Future Work

In this paper, 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. 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 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 used 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.

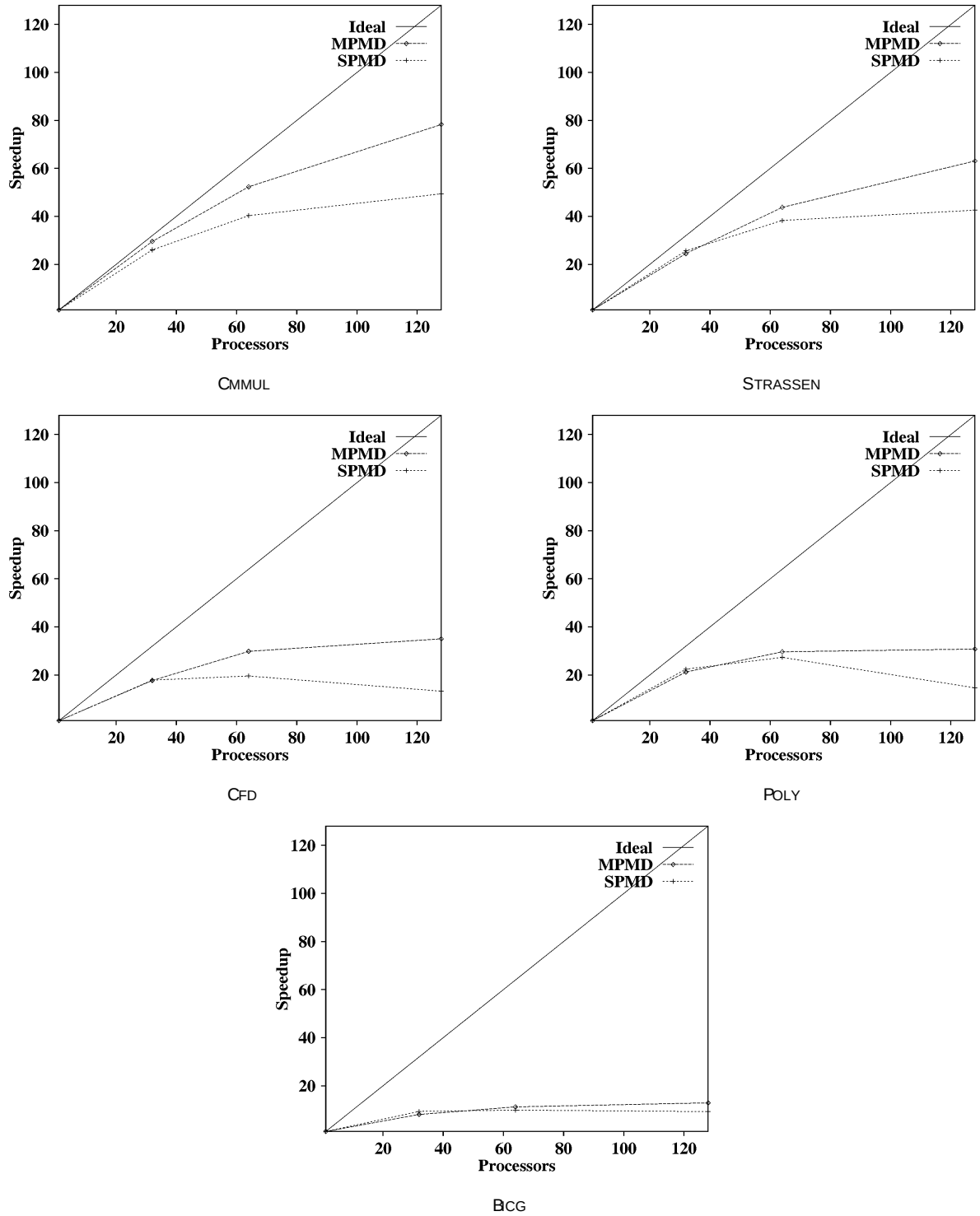


Fig. 15. Speedup measurements for benchmark applications on the CM-5.

An interesting issue we would like to consider in the future is the use of 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 is the investigation of techniques for exploiting task and data parallelism in irregular scientific applications [48]. These applications may require dynamic run-time allocation and scheduling techniques in contrast to the static techniques used in this paper.

ACKNOWLEDGMENTS

This research was supported in part by an IBM Graduate Fellowship and in part by the National Aeronautics and Space Administration under contract NASA NAG 1-613.

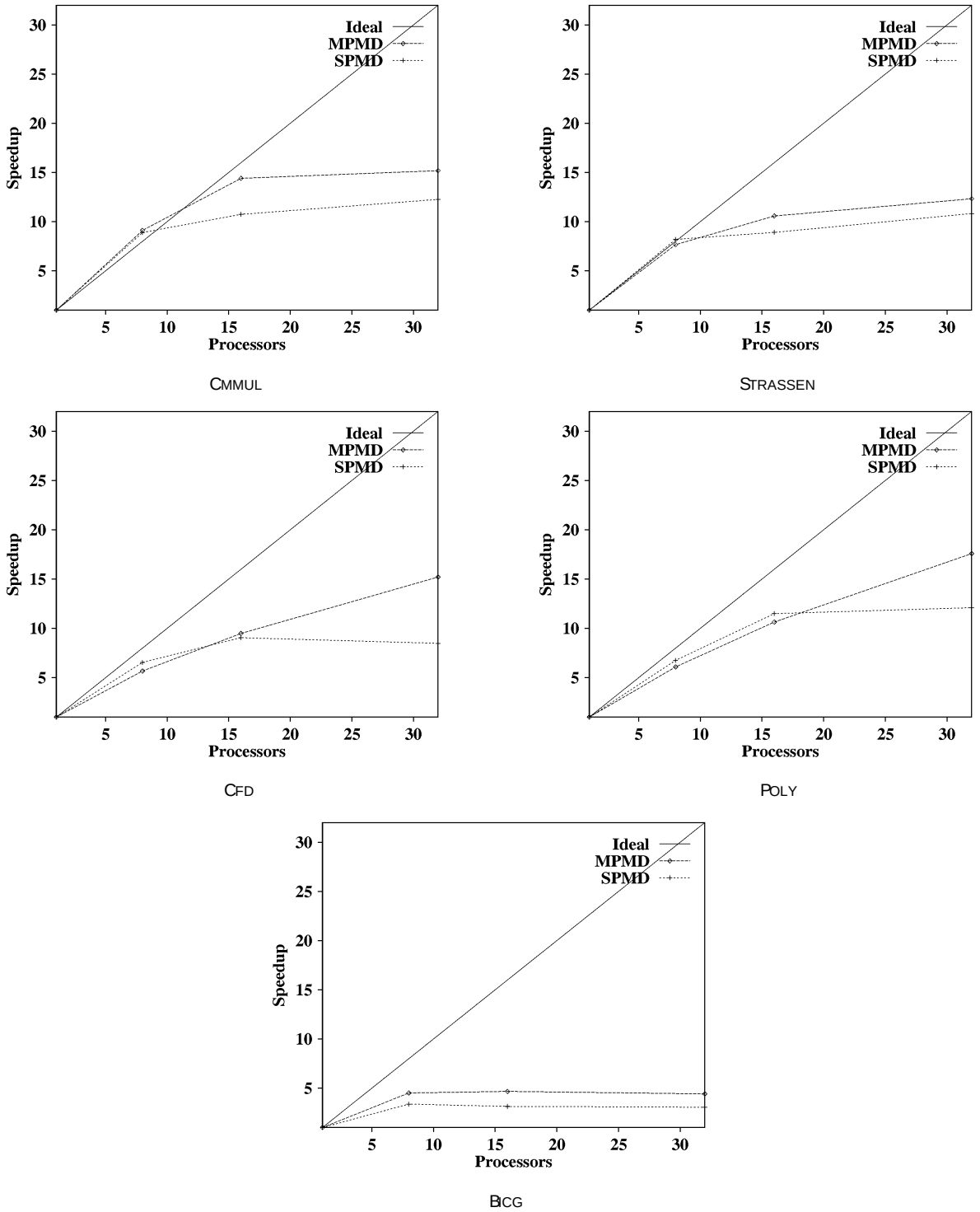


Fig. 16. Speedup measurements for benchmark applications on the Paragon.

REFERENCES

[1] High Performance Fortran Forum, "High Performance Fortran Language Specification, version 1.1," technical report, Center for Research on Parallel Computation, Rice Univ., Houston, Texas, Nov. 1994.

[2] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel, *The High Performance Fortran Handbook*. Cambridge, Mass.: MIT Press, 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," *Computer*, vol. 28, no. 10, pp. 37–47, Oct. 1995.

[4] S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," *Proc. Frontiers '95: The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pp. 342–349, McLean, Va., Feb. 1995.

[5] S. Ramaswamy, "Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications," PhD thesis CRHC-96-03/UILU-ENG-96-2203, Dept. of Electrical and Computer Eng., Univ. of Illinois, Urbana, Jan. 1996.

[6] I. Foster, B. Avalani, A. Choudhary, and M. Xu, "A Compilation System That Integrates High Performance Fortran and Fortran M," *Proc. Scalable High Performance Computing Conf.*, pp. 293–300, Knoxville, Tenn., May 1994.

[7] 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.

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

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

[10] 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," *Proc. Supercomputing*, San Diego, Calif., Dec. 1995.

[11] *Digital High Performance Fortran 90 HPF and PSE Manual.* Maynard, Mass.: Digital Equipment Corp., 1995.

[12] *PGHPF User's Guide*. Wilsonville, Ore.: Portland Group Inc., 1995.

[13] *XHPF User's Guide, Version 2.0.* Placerville, Calif.: Applied Parallel Research, 1995.

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

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

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

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

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

[19] 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," *Proc. 18th Int'l Conf. Parallel Processing*, pp. 39–48, St. Charles, Ill., Aug. 1989.

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

[21] M. Girkar, "Functional Parallelism: Theoretical Foundations and Implementations," PhD thesis CSRD-1182, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, Dec. 1991.

[22] J.E. Moreira, "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors," PhD thesis, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, Jan. 1995.

[23] M. Dhagat, R. Bagrodia, and M. Chandy, "Integrating Task and Data Parallelism in UC," *Proc. Int'l Conf. Parallel Processing*, pp. 29–36, Oconomowoc, Wis., Aug. 1995.

[24] 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.

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

[26] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, Mass.: MIT Press, 1989.

[27] T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," *Proc. Supercomputing*, pp. 633–642, Albuquerque, N.M., Nov. 1991.

[28] T. Yang and A. Gerasoulis, "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors," *Proc. Scalable High Performance Computing Conference*, pp. 350–357, Williamsburg, Va., Apr. 1992.

[29] G.N.S. Prasanna and A. Agarwal, "Compile-Time Techniques for Processor Allocation in Macro Dataflow Graphs for Multiprocessors," *Proc. Int'l Conf. Parallel Processing*, pp. 279–283, St. Charles, Ill., Aug. 1992.

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

[31] K.P. Belkhale and P. Banerjee, "Approximate Algorithms for the Partitionable Independent Task Scheduling Problem," *Proc. 19th Int'l Conf. Parallel Processing*, pp. 72–75, St. Charles, Ill., Aug. 1990.

[32] K.P. Belkhale and P. Banerjee, "A Scheduling Algorithm for Parallelizable Dependent Tasks," *Proc. Int'l Parallel Processing Symp.*, pp. 500–506, Anaheim, Calif., Apr. 1991.

[33] 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," *Proc. Supercomputing '94*, pp. 330–339, Washington D.C., Nov. 1994.

[34] J. Subhlok and G. Vondran, "Optimal Mapping of Sequences of Data Parallel Tasks," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 134–143, Santa Barbara, Calif., July 1995.

[35] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten, "Parafrase-2 Manual," technical report, Center for Supercomputing Research and Development, Univ. of Illinois, Urbana, Aug. 1990.

[36] E.W. Hodges IV, "High Performance Fortran Support for the PARADIGM Compiler," MS thesis CRHC-95-23/UILU-ENG-95-2237, Dept. of Electrical and Computer Eng., Univ. of Illinois, Urbana, Oct. 1995.

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

[38] P.M. Vaidya, "A New Algorithm for Minimizing Convex Functions Over Convex Sets," *Proc. Symp. Foundations of Computer Science*, pp. 332–337, Research Triangle Park, N.C., Oct. 1989.

[39] M. Gupta, "Automatic Data Partitioning on Distributed Memory Multicomputers," PhD thesis CRHC-92-19/UILU-ENG-92-2237, Dept. of Computer Science, Univ. of Illinois, Urbana, Sept. 1992.

[40] M. Gupta and P. Banerjee, "Compile-Time Estimation of Communication Costs on Multicomputers," *Proc. Sixth Int'l Parallel Processing Symp.*, pp. 470–475, Beverly Hills, Calif., Mar. 1992.

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

[42] 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.

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

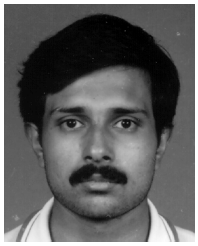
[44] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," *Proc. 23rd Int'l Conf. Parallel Processing*, vol. II, pp. 116–125, St. Charles, Ill., Aug. 1994.

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

[46] S.L. Lyons, T.J. Hanratty, and J.B. McLaughlin, "Large-Scale Computer Simulation of Fully Developed Channel Flow with Heat Transfer," *Int'l J. Numerical Methods for Fluids*, vol. 13, no. 8, pp. 999–1,028, Nov. 1991.

[47] 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.

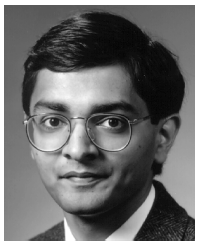
[48] A. Lain, "Compiler and Run-Time Support for Irregular Computations," PhD thesis CRHC-92-22, Dept. of Computer Science, Univ. of Illinois, Urbana, Oct. 1995.



Shankar Ramaswamy received the BSc degree in electronics from Delhi University, New Delhi, India, in July 1987, the ME degree in electrical engineering from the Indian Institute of Science, Bangalore, India, in July 1991, and the PhD degree in electrical engineering from the University of Illinois at Urbana-Champaign in May 1996.

Dr. Ramaswamy is currently a member of the technical staff at Transarc Corp., Pittsburgh, Pennsylvania.

Dr. Ramaswamy's research interests are in the area of systems software for Distributed and Parallel computer systems. He was the receipient of an IBM Graduate fellowship award while at the University of Illinois, the Alfred Hay gold medal from the Indian Institute of Science, and the Smt. Prakashwati Memorial Award from Delhi University.

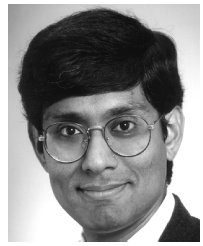


Sachin Sapatnekar received the BTech degree from the Indian Institute of Technology, Bombay, in 1987, the MS degree from Syracuse University in 1989, and the PhD degree from the University of Illinois at Urbana-Champaign in 1992.

Dr. Sapatnekar is currently an associate professor in the Department of Electrical Engineering at the University of Minnesota. From 1992 to 1997, he was an assistant professor in the Department of Electrical and Computer Engineering at Iowa State University. He also worked at Texas Instruments Inc., Dallas, Texas, in 1990, and at Intel Corporation, Santa Clara, California, in 1997.

Dr. Sapatnekar's research interests lie in developing efficient techniques for computer-aided design of integrated circuits and are primarily centered around physical design, power, timing and simulation issues, and optimization algorithms. He has authored several papers in this area, as well as a book entitled *Design Automation for Timing-Driven Layout Synthesis*, published by Kluwer Academic Publishers, Boston, Massachusetts. Dr. Sapatnekar served as an associate editor for *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, and served on program committees for several conferences.

Dr. Sapatnekar is a recipient of the National Science Foundation Career Award and a Best Paper award at the 1997 Design Automation Conference.



Prithviraj Banerjee received the BTech degree in electronics and electrical engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981, and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign in 1982 and 1984, respectively.

Dr. Banerjee is currently the Walter P. Murphy chaired professor of Electrical and Computer Engineering and director of the Center for Parallel and Distributed Computing at Northwestern University in Evanston, Illinois. Prior to that, he was the director of the Computational Science and Engineering Program, and a professor in the Electrical and Computer Engineering Department and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign.

Dr. Banerjee's research interests are in parallel algorithms for VLSI design automation, distributed memory parallel compilers, and parallel architectures with an emphasis on fault tolerance. He is the author of more than 180 papers in these areas. He leads the PARADIGM compiler project for compiling programs for distributed memory multicomputers, and ProperCAD project for portable parallel VLSI CAD applications. He is also the author of a book entitled *Parallel Algorithms for VLSI CAD* published by Prentice Hall, in 1994. He has supervised 22 PhD and 25 MS student theses thus far.

Dr. Banerjee has received numerous awards and honors during his career. He is the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division sponsored by Hewlett-Packard. He was elected a fellow of the IEEE in 1995. He received the University Scholar award from the University of Illinois for in 1993, the Senior Xerox Research Award in 1992, IEEE senior membership in 1990, the National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981.

Dr. Banerjee served as the program chair of the International Conference on Parallel Processing for 1995. He served on the program and organizing committees of the 1988, 1989, 1993, and 1996 Fault Tolerant Computing Symposia, the 1992, 1994, 1995, 1996, and 1997 International Parallel Processing Symposia, the 1991, 1992, and 1994 International Symposia on Computer Architecture, the 1990, 1993, 1994, 1995, 1996, and 1997 International Symposia on VLSI Design, the 1994, 1995 and 1996 International Conference on Parallel Processing, and the 1995 and 1996 International Conference on High-Performance Computing. He also served as general chairman of the International Workshop on Hardware Fault Tolerance in Multiprocessors, 1989. He is an associate editor of the *Journal of Parallel and Distributed Computing* and *IEEE Transactions on Computers*. In the past, he has served as editor of the *IEEE Transactions of VLSI Systems*, and the *Journal of Circuits, Systems, and Computers*.