A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers *

Shankar Ramaswamy†, *Sachin Sapatnekar*‡ *and Prithviraj Banerjee*†

† Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801

‡ Dept of EE/CprE
222 Coover Hall
Iowa State University
Ames, IA 50011

Tel : (217)333-6564
Fax : (217)244-5685
E-mail : shankar,banerjee@crhc.uiuc.edu

Abstract Recent research efforts have shown the benefits of integrating functional and data parallelism over using either pure data parallelism or pure functional parallelism. The work in this paper presents a theoretical framework for deciding on a good execution strategy for a given program based on the available functional and data parallelism in the program. The framework is based on assumptions about the form of computation and communication cost functions for multicomputer systems. We present mathematical functions for these costs and show that these functions are realistic. The framework also requires specification of the available functional and data parallelism for a given problem. For this purpose, we have developed a graphical programming tool. Currently, we have tested our approach using three benchmark programs on the Thinking Machines CM-5 and Intel Paragon. Results presented show that the approach is very effective and can provide a two- to three-fold increase in speedups over approaches using only data parallelism.

Keywords Functional and Data Parallelism, Processor Allocation, Parallel Task Scheduling, Macro Dataflow Graphs, Graphical Programming, Cost Models, Distributed Memory Multicomputers, Convex Programming

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

1 Introduction

Distributed Memory Multicomputers such as the IBM SP-1, the Intel Paragon and the Thinking Machines CM-5 offer significant advantages over shared memory multiprocessors in terms of cost and scalability. Unfortunately, to extract all that computational power from these machines, users have to write efficient software for them, which is an extremely laborious process. Numerous research efforts have proposed language extensions to FORTRAN in order to ease programming multicomputers; the most prominent one has been the HPF language standardization [1]. A number of compilers for HPF have been proposed; these include the FORTRAN-D compiler from Rice University [2], the SUIF compiler from Stanford [3], the PTRAN II compiler from IBM [4], the SUPERB compiler from the University of Vienna [5], and, the FORTRAN-90D/HPF compiler from Syracuse University [6].

The PARADIGM compiler project at Illinois is aimed at devising a parallelizing compiler for distributed memory multicomputers that will accept FORTRAN 77 or HPF programs as input. The fully implemented PARADIGM compiler will:

- Annotate FORTRAN 77 programs with HPF data distribution directives [7, 8].
- Partition computations and generate communication for HPF programs [9, 10, 11].
- Exploit functional and data parallelism in HPF programs [12, 13, 14, 15].
- Provide runtime support for irregular computations [16].

There has been a lot of interest in simultaneous exploitation of data and functional parallelism. Research efforts in the area include the Fx compiler from CMU [17], the FORTRAN-M compiler from Argonne National Lab [18], the work by Chapman et. al. in [19], the work by Cheung and Reeves in [20], the work by Girkar and Polychronopoulos in [21], and, the work by Ramaswamy and Banerjee in [22]. All these efforts recognize the benefits of using both types of parallelism together to achieve better performance for certain applications. In this paper, we have discussed the framework to be used in the PARADIGM compiler for exploiting functional and data parallelism together.

For our discussion, we define *Functional Parallelism* to be any parallelism existent among the various routines of a given program and *Data Parallelism* to be parallelism within a routine that is obtained by distributing data among all processors involved and having them each perform computation using the owner computes rule [2]. Matrix Add, Matrix Multiply and 2D FFT are a few examples of what we mean by routines. To re-emphasize, the definitions of functional and data parallelism above may not correspond to some of the other popular connotations of these terms.

1.1 Macro Dataflow Graphs

In order to expose the parallelism available in any given program, we use a representation called the *Macro Dataflow Graph* (MDG). This representation has been used before by researchers such as Sarkar in [23] and Prasanna and Agarwal in [24]. For our work, the MDG representation for a program is defined to be a weighted directed acyclic graph whose nodes correspond to routines of the program and edges correspond to precedence constraints among these routines. There are two distinguished nodes in the MDG called START and STOP. START precedes all other nodes and STOP succeeds all other nodes.

The weights of the nodes and edges are based on the concepts of *Processing* and *Data Transfer* costs. The time required for the execution of a routine is called its processing cost. Processing costs will depend on the number of processors used to execute the routine and include all computation and communication costs incurred. On distributed memory machines these costs will be dependent on the kind of data distribution used. Each routine may need a particular distribution for each of its arrays to achieve the best performance. Since precedence constraints mean that an array being read by a routine is being written into by its predecessor, we may need to redistribute the array after the execution of the predecessor routine. The time needed for this data redistribution between the execution of a pair of routines is referred to as the data transfer cost for that pair. Data transfer costs are made up of three components : a sending cost for processors at the sending routine, a network cost, and, a receiving cost for processors at the receiving routine. All these cost components are functions of the number of processors used for the sending and receiving routines.

We consider the weight of a node in the MDG to be composed of:

1. The receiving cost components of all data transfers from its predecessors

2. The processing cost of the routine it corresponds to

3. The sending cost components of all data transfers to its successors

The two distinguished nodes START and STOP do not perform any computation, they have zero weight.

The weight of an edge between a pair of nodes in the MDG is taken to be the network cost component of data transfer between the routines corresponding to the nodes.

The usefulness of MDGs is that they can be used to decide on the strategy to be used to minimize execution time of the given program on the target multicomputer. MDGs expose functional and data parallelism in the program, allowing us to exploit both in an optimal manner. Data parallelism information is implicit in the weight functions of the nodes and functional parallelism is implicit

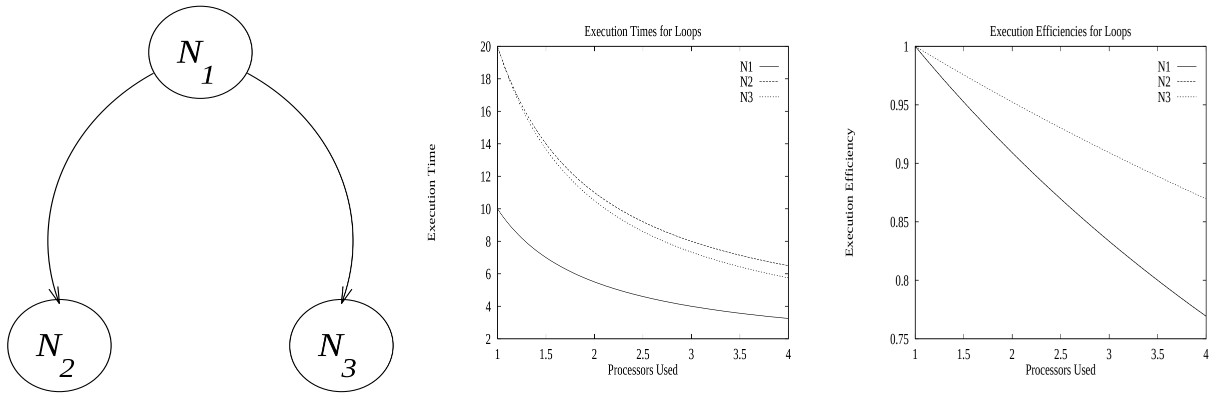


Figure 1: Example Showing Functional Parallelism

in the precedence constraints among nodes. In order to decide on a good execution strategy for a program, we use an *Allocation and Scheduling* approach. Allocation decides on the number of processors to use for each node in the MDG and scheduling decides on a scheme of execution for the allocated nodes on the target multicomputer. Our work in this paper provides methods that allocate and schedule any given MDG such that the finish time obtained is within a factor of the best finish time theoretically obtainable.

1.2 Example

The usefulness of good allocation and scheduling may not be clear at first sight. It can be better appreciated by considering an example. Figure 1 shows an MDG with three nodes N_1, N_2 and N_3. Plotted alongside are the processing costs of the routines they correspond to as a function of the number of processors used. For ease of understanding we assume there are no data transfer costs between routines. By our definitions, the weights of the nodes in this MDG would be the same as the corresponding processing costs and the weight of edges would be 0. Now, given a system with 4 processors, there could be many ways in which we can allocate and schedule the MDG. For instance, a naive scheme would be to execute the nodes one after another on all 4 processors. In this case, we have an execution time of 15.6 seconds. On the other hand, a better way of executing the MDG would be to first execute N_1 on all 4 processors, then allocate 2 processors each to nodes N_2 and N_3 and execute them concurrently. This way, the routines finish in 14.3 seconds. The two schemes are shown pictorially in Figure 2. The first scheme is exploiting pure data parallelism, i.e., all routines use 4 processors. The second scheme on the other hand, is exploiting both functional and data parallelism, i.e., routines 2 and 3 execute concurrently as well as use 2 processors each.

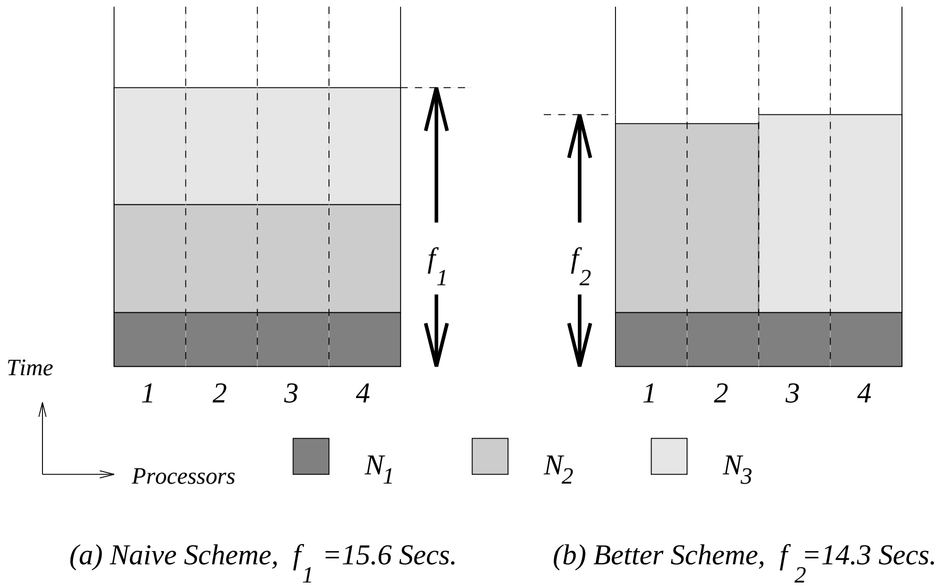


Figure 2: Allocation and Scheduling Schemes for Example

Intuitively, good allocation and scheduling makes program execution faster because of more efficient execution. Most real applications execute more inefficiently as the size of a processor system grows, the processing efficiency curves of Figure 1 in our example are typical. We can see that by executing the nodes N_2 and N_3 concurrently and using fewer processors for them, the second scheme improves overall efficiency over the first. This makes the second scheme execute the program faster than the first.

A point of interest with respect to the type of code generated in the two schemes is that the first scheme will essentially have each processor execute similar code on different data sets. We refer to this type of code as Single Program Multiple Data (SPMD). On the other hand, the second scheme can have very different code for each processor; this type of code is called Multiple Program Multiple Data (MPMD). Therefore, SPMD code exploits only data parallelism while MPMD code exploits both data and functional parallelism.

1.3 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 in [25]. Further treatment on this topic can also be found in the book by Garey and Johnson [26]. The allocation and scheduling problem is considerably harder than the one just described. 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 in [23], and Gerasoulis and Yang in [27, 28]. A bottom up approach considers the MDG to be made up of lightweight nodes (in terms of computation requirements) with each using only one processor (an explicit allocation is not done). The bottom up scheduling methods of [23, 27, 28] then use clustering on the nodes to form larger nodes during the construction of a schedule. The second approach to allocation and scheduling is a top down approach like the ones used by Prasanna and Agarwal in [24], Belkhale and Banerjee in [12, 13], by Subhlok et. al. in [17, 29, 30], by Ramaswamy and Banerjee in [14, 15] and in this paper. Top down approaches start with the assumption of heavyweight nodes (again, in terms of computation requirements) in the MDG and break them down during the process of constructing an optimal schedule. Top down methods are considered better in that they take a more global view of the problem than the bottom up approaches. Therefore they may be able to perform better optimizations.

The difference between earlier top down approaches mentioned above and the work presented here is significant. The methods presented in [24] do not consider data transfer costs between nodes of the MDG. In addition, they 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 [12, 13] also does not consider the effects of non-zero data transfer costs. Their allocation and scheduling algorithms are similar to the ones we use. The research presented in [17, 29, 30] considers allocation and scheduling for a class of problems that process continuous streams of data sets. The computation for each data set has a tree-structured MDG for all their benchmark programs [31]. A set of heuristics are used to decide on a good allocation and scheduling scheme. There is no performance analysis of these heuristics and it is not clear how they would work for more general MDGs (DAGs). Our methods on the other hand have been theoretically analyzed for performance bounds and work well for general MDGs as we will show.

1.4 MAST : MDG Allocation and Scheduling Tool

In order to provide an interface to our MDG allocation and scheduling methods, we designed and implemented MAST. Some of the ideas used for MAST are similar to the ones used for the HeNCE tool [32]. Basically, MAST provides users with the capability of specifying the MDG representation for their programs in a graphical manner. Once an MDG has been specified, MAST helps the user study the performance of his code on various architectures and run the code if needed on any of those architectures.

MAST has three major components to it:

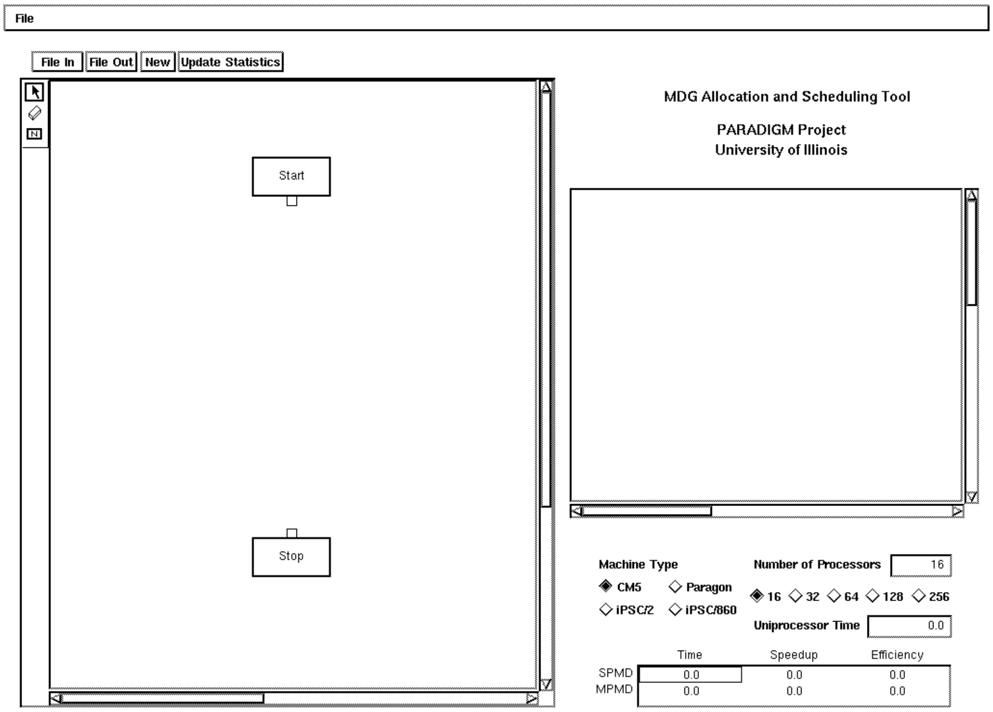


Figure 3: Startup View of MAST

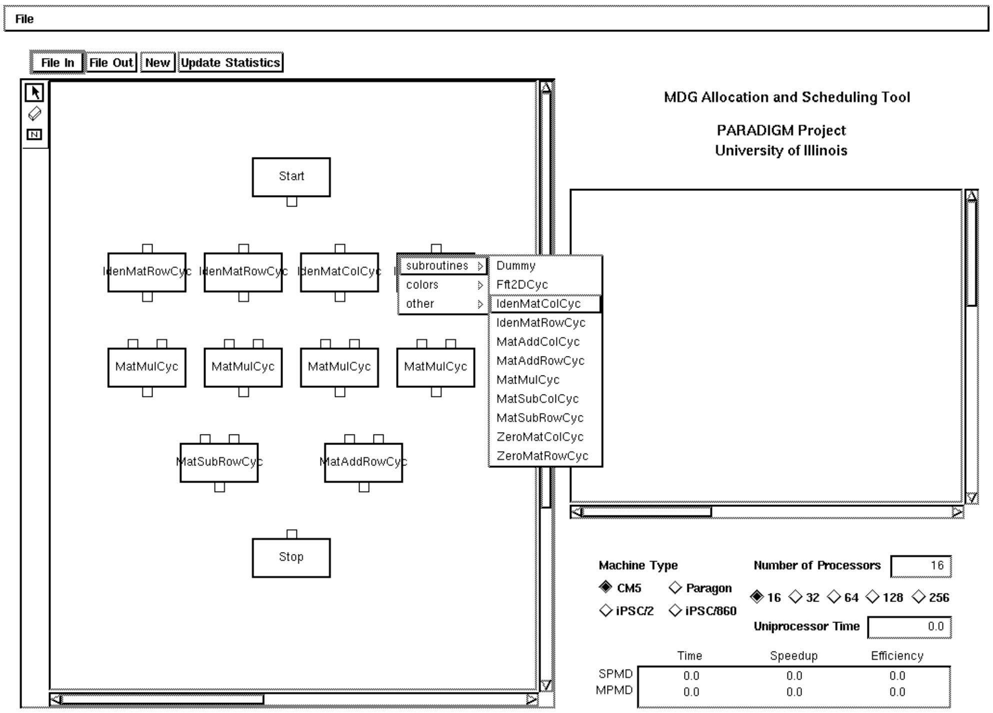


Figure 4: View of MAST After Nodes Have Been Drawn

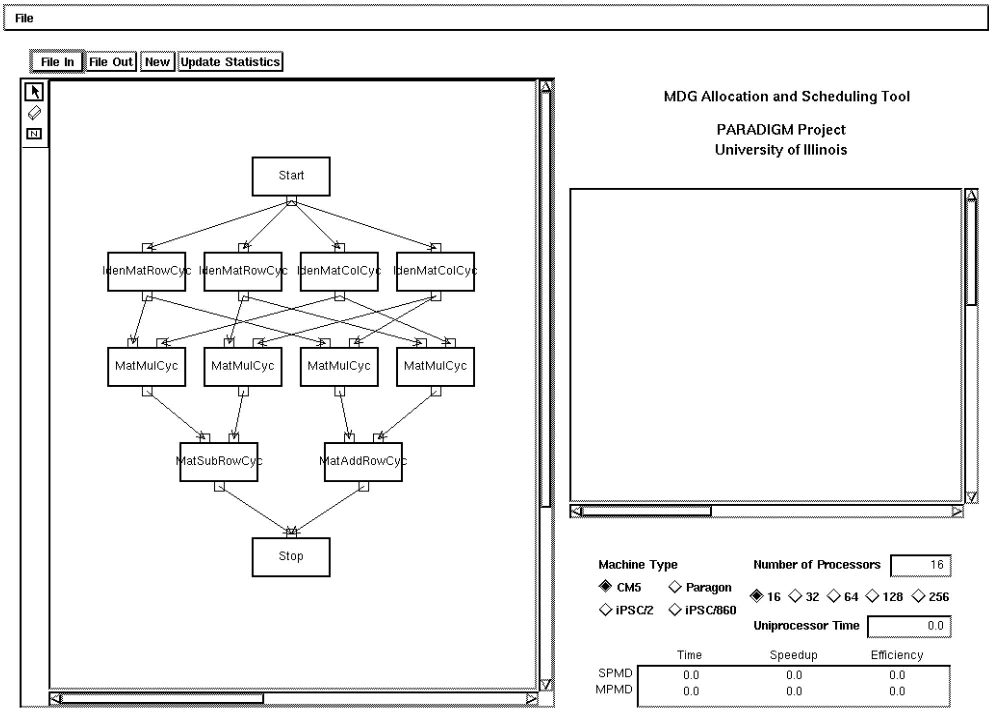


Figure 5: View of MAST After a Complete MDG Has Been Drawn

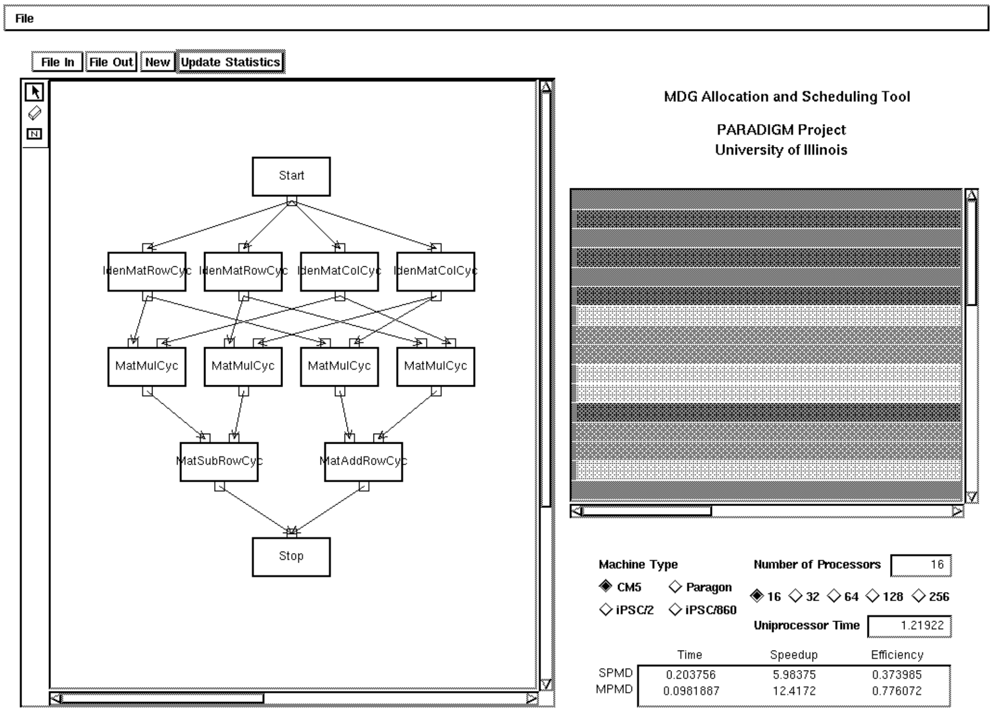


Figure 6: View of MAST After an Update Statistics

1. A graphical programming tool
2. A library of parallel scientific routines whose execution is well profiled on all the desired target multicomputers
3. An allocation and scheduling tool based on the methods discussed in this paper

MAST ties up the three components and provides the user with various utilities. A step by step use of MAST has been shown in Figures 3 through 6. We now explain the significance of each figure:

Figure 3 shows MAST when it is started up. At this point the graphical programming tool on the left half of MAST has only two nodes – START and STOP.

Figure 4 shows MAST after the user has decided on the routines to be used in his program and placed the required nodes. Nodes are drawn using the one of the utilities of the graphical programming tool that can be seen on the top left corner of the canvas. Once a node has been drawn, it can be tied to a library routine using a pull down menu provided (shown in figure). On closer inspection of the figure, the nodes can be seen to have different routine names on them. It can also be seen that each node has little tag boxes on top and bottom; these represent the input and output arrays for the routine the node represents. Different routines have different numbers of tag boxes depending on their input and output array counts.

Figure 5 shows MAST after the user has connected the nodes using the edge drawing utility of the graphical programming tool. Edges connect an output tag box of a node to an input tag box of another node. This indicates the array being written into by the first node is being read by the second node. Output tag boxes may have multiple edges, input tag boxes can have only one edge.

Figure 6 shows MAST after the user has completed the MDG and has asked for a performance evaluation on a specified target architecture of a specified size. This performance evaluation uses the execution profile information of the scientific library. Performance statistics provided include predicted uniprocessor time, SPMD time and MPMD time. Speedup and efficiency predictions are also provided for the SPMD and MPMD cases; also provided is a Gantt chart showing the allocation and scheduling used for the MPMD case. In addition, MAST uses the allocation and schedule computed to generate source code containing:

- Calls to routines in the scientific library provided in MAST.

- Routines generated for data redistribution – this is done using the work discussed in [22]. In that paper, techniques for redistributing arrays (for regular distributions) between arbitrary processor sets have been discussed.
- Routines generated to enable the scientific routines and redistribution routines to execute on subsets of processors. These routines are based on concepts similar to those of groups, contexts and communicators in MPI [33].

This generated code is ready to be compiled and executed on the target architecture.

In contrast to our graphical programming approach, other researchers in the area of integrating data and functional parallelism have used language extensions for specifying available data and functional parallelism. The work in [17, 29, 30] on the Fx compiler is based on extensions of FORTRAN which are used to specify functional and data parallelism. Data parallelism is specified using constructs similar to HPF and functional parallelism is specified using constructs called Parallel Sections. The FORTRAN-M language inherently provides constructs for specifying functional parallelism [18]; recent work proposes to integrate the language with HPF in order to specify data parallelism [34]. The work in [19] has proposed extensions to FORTRAN-90 for specifying functional parallelism.

In the next section, we provide a brief overview of the theory of convex and posynomial functions which we use in developing our allocation algorithm. In the following sections we discuss our allocation and scheduling algorithms. We then present our processing and data transfer cost models in Section 5. Theoretical results that discuss the optimality of our algorithms are provided in Section 6. Section 7 provides preliminary results obtained using our algorithms.

2 Theory of Convex and Posynomial Functions

In this section 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 [35]. A discussion of the theory of posynomial functions is provided by Ecker in [36]. We have selected a few important and relevant points about these functions for our discussion here.

2.1 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 $\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2 \in C$.

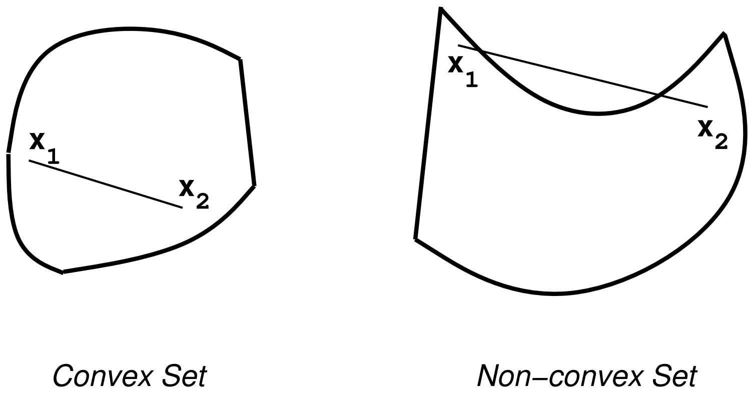


Figure 7: Convex sets.

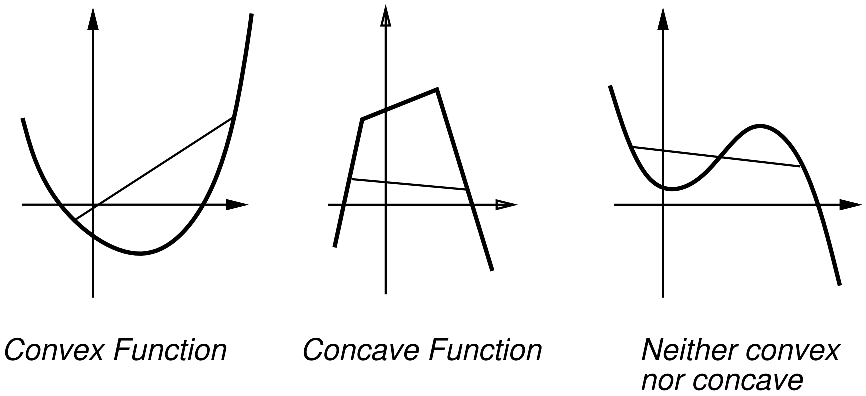


Figure 8: Convex Functions

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 7.

2.2 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, $0 \leq \alpha \leq 1$,

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

f is said to be *strictly convex* if the inequality in Equation (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 Fig 8.

2.3 The Convex Programming Problem

The convex programming problem is stated as follows:

$$\textit{minimize} \quad f(\mathbf{x}) \tag{2}$$

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

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.

2.4 Posynomial Functions

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

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

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 [36]

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

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.

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:

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

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

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

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

The fact that these functions are posynomials will be used later in the paper.

2.5 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 functions $f + g$ is a convex function over S.

Constant Property The function $c \cdot f$, where c is a non-negative 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 5. Therefore, given two posynomial functions h and j defined on S, all the above properties hold for the pair. We will use these properties later in the paper.

3 MDG Allocation Algorithm

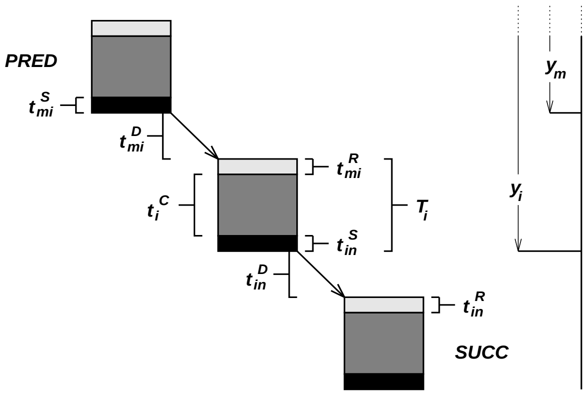
We first consider the problem of allocation of processors to the nodes of a given MDG. After the allocation is carried out using this algorithm, the MDG is ready to be scheduled using the algorithm described in the next section.

For the purposes of allocation and scheduling, we assume the given MDG has n nodes numbered consecutively from 1 to n. In addition, node 1 is the START node and node n is the STOP node.

To obtain an optimum solution to the allocation problem for a given MDG and a given p processor target system, we solve:

minimize Φ, where:

$$\Phi = \max(A_p, C_p)$$
$$A_p = \frac{1}{p} \cdot \sum_{i=1}^{n} T_i \cdot p_i$$
$$C_p = y_n$$
$$y_i = \max_{m \in PRED_i}(y_m + t_{mi}^D) + T_i$$
$$T_i = (\sum_{m \in PRED_i} t_{mi}^R + t_i^C + \sum_{n \in SUCC_i} t_{in}^S)$$



where

1. p_i represents the number of processors used by the ith node.

2. t_i^C is the processing cost of the routine corresponding to node i and is a function of p_i.

3. t_{mi}^R represents the time required at node i to process the messages it receives from predecessor node m (receiving cost component of data transfer). t_{mi}^D represents the network delay required between the completion of node i and the start of node m (network cost component of data transfer, weight of edge between nodes m and i). t_{in}^R represents the time required at node i to process messages it sends to successor node n (sending cost component of data transfer). All these quantities are functions of p_i and p_j.

4. $PRED_i$ and $SUCC_i$ are the sets of predecessor and successor nodes of node i in the given MDG, respectively.

5. T_i is the total time required to process node i (weight of node i).

6. y_i is the finish time of the ith node.

7. Φ is the *Optimum Finish Time* obtainable for the execution of the program corresponding to the given MDG.

8. A_p is also called the *Average Finish Time* for the case when nodes use up to p processors each. To better understand the idea behind using the average finish time, consider a quantity called processor-time area for a node. This is the product of time taken for executing a node and the number of processors it uses. If we sum the processor-time areas for all nodes in the MDG, this will represent the minimum processor-time area requirement for the MDG. Another way of saying the same is that Φ must be at least same as the average finish time which represents the sum of processor-time areas of all the nodes in the MDG averaged over p.

9. C_p is called the *Critical Path Time* for the case when nodes use up to p processors each. Since the critical path is the longest in the MDG, it represents the shortest possible time in which we can finish executing the MDG. This implies Φ must be at least same as the critical path time.

The free variables in this formulation are the p_i's, with $1 \leq p_i \leq p, \; i = 1, n$.

Our formulation relies on the properties of convex functions and posynomial functions discussed in the previous section. Basically, our allocation problem is equivalent to a convex programming formulation if the following conditions hold:

1. t_{ij}^D, t_{ij}^R, t_{ij}^S, and t_i^C can all be represented by posynomial functions of the free variables.

2. $t^R_{ij} \cdot p_j$, $t^S_{ij} \cdot p_i$ and $t^C_i \cdot p_i$ are also posynomial functions of the free variables.

Later, in Section 5, we present cost functions to represent the quantities t^D_{ij}, t^R_{ij}, t^S_{ij}, and t^C_i which satisfy these conditions. We also demonstrate the practicality of these functions.

The discussion above implies that in practice, we can construct a formulation equivalent to a convex programming formulation for allocation, and, therefore, obtain a unique minimum value for Φ. The allocation that corresponds to this value will be an optimum allocation for the given MDG. This method of allocation inherently assumes the existence of a perfect scheduler, i.e. one that can produce a schedule which finishes the program in Φ time units. In practice, producing such a schedule is an NP-Complete problem [26]. We therefore, use a scheduler as described in the next section which might produce a finish time different from Φ. As we will show in Section 6, we have quantified this deviation.

4 MDG Scheduling Algorithm

To schedule a given MDG with processor allocation done according to the method described in the previous section, we use an algorithm called the *Prioritized Scheduling Algorithm* (PSA). The steps involved in the PSA are:

1. The processor allocation produced by the convex programming formulation will be a set of positive real numbers in the general case, however, we cannot allocate processors in this manner on a real system. In this step we round-off the allocated processors for all the nodes to the nearest power of 2. This is done to make the final code generation very easy. The results we obtain in Section 7 will show that this does not result in much loss in practice. We refer to this step in the sections that follow as the rounding-off step.

2. The rounded-off processor allocation for the MDG is then modified to impose a bound (PB) on the number of processors used by any node. If the ith node uses p_i processors and $p_i > PB$, p_i is reduced to PB, else it is left unchanged. It can be seen that PB has to be a power of two or else we will have to round-off again and that may lead to a violation of the bound. The value of PB to be used is determined using Theorem 3 which is discussed in Section 6. We refer to this step in the sections that follow as the bounding step.

3. Since the processor allocation for the MDG may have been changed from the value produced by the allocation step, we need to re-compute the weights of the nodes and the edges of the MDG based on the new allocation. Next, we place the node START on a queue called the ready queue and mark its Earliest Start Time (EST) as 0.

4. In this step, we pick a node from the ready queue that has the lowest possible EST. We then check to see the time at which the processor requirement of this node can be met, i.e., the time at which the required processors will be done with the node(s) they are currently processing and can accept another node. This is called the Processor Satisfaction Time (PST). If $PST \geq EST$, the node can be scheduled at PST else, it can be scheduled only at EST. It must be noted that there will be some idle time in the latter case since the required processors are available but not used. However, the scheduler is not forcing idleness, it simply does not have any other node to schedule since we have picked the node with the lowest EST.

5. If the node just scheduled is the STOP node, the scheduler is terminated else, we go to the next step.

6. After scheduling the node, we now check to see if any of its successors have all their predecessors scheduled, i.e. have their precedence constraints satisfied. If so, we compute the EST for those nodes based on the node and edge weights of the MDG and the schedule built so far. Such nodes are then placed in the ready queue.

7. Steps are repeated starting at Step 4.

The finish time of the STOP node based on the schedule is the predicted finish time of the program.

The scheduling algorithm described above is a variant of the popular List Scheduling Algorithm (LSA) which has been used for example, by Liu in [37], by Garey, Graham and Johnson in [38], by Wang and Cheng in [39], by Belkhale and Banerjee in [13], by Turek, Wolf and Yu in [40], and, by Ramaswamy and Banerjee in [14, 15]. It must be noted that some of the mentioned researchers also use variants of the LSA. We call it the PSA because of the implicit prioritization in Step 4 where a node with the lowest EST is picked even though other nodes may be ready for scheduling.

In the case where the number of processors used by any node is bounded, the PSA is shown to be within a factor of the optimum in Theorem 1 in Section 6. While similar results have been shown in the references mentioned above when there are no data transfer costs, our result is unique in that it takes into account these costs. In fact, it is the first such result to be derived.

5 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 satisfy two criteria; they have to be convex or posynomial functions, and, they have to be practical.

In this section we show that our cost functions are posynomials, their suitability is shown in Section 7.

The processing cost function we use is an often used model. On the other hand, the data transfer cost functions are our own. The derivation of these functions is described in detail in [41].

Processing Cost Model

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

$$t_i^C = (\alpha_i + \frac{1-\alpha_i}{p_i}) \cdot \tau_i \tag{10}$$

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

$$0 \leq \alpha_i \leq 1$$

$$0 \leq \tau_i \tag{11}$$

The way we calculated *alpha* and *tau* for the various routines used in our benchmarks is by actually measuring execution times for these routines 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 we have chosen. In the future, we are considering the use of static techniques to predict these values. At this point, we only wish to demonstrate that processing costs can be modeled by a function of the form shown above. As our results will show, our form models processing costs fairly accurately in practice.

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

Proof : From Equation 10 we can 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 non-negative constant (under Equation 11). The second component has a non-negative constant factor multiplying a posynomial $\frac{1}{p_i}$. Under the Constant Property discussed in Section 2, this component is also a posynomial. Since both components are posynomials, using the Sum Property of Section 2, t_i^C is a posynomial. □

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

Proof : Using Equation 10, we can write down:

$$t_i^C \cdot p_i = \alpha_i \cdot \tau_i \cdot p_i + (1 - \alpha_i) \cdot \tau_i \tag{12}$$

We can see that this equation has two components; a variable component $\alpha_i \cdot \tau_i \cdot p_i$ which has a non-negative constant factor $\alpha_i \cdot \tau_i$ multiplying a posynomial p_i. By the Constant Property of Section 2, this component is a posynomial. The other component in the equation above is a non-negative constant which is also a posynomial. Hence, using the Sum Property of Section 2, we see that $t_i^C \cdot p_i$ is a posynomial. □

We would like to point out that α and τ for a routine could depend on the size of data input to the routine. This does not affect the statements made in either Lemma above.

Data Transfer Cost Model

Here we consider the cost of redistribution of an array of data elements between the execution of two nodes of the MDG involving p_i and p_j processors at the sending and receiving ends respectively. For modeling such a transfer, we assume that the array is distributed evenly across the p_i sending processors initially, and across the p_j receiving processors finally. In addition, we assume that the number and sizes of messages will be same for each sending processor and for each receiving processor. For example, every sending processor may send 3 messages of 1000 bytes and every receiver may receive 5 messages of 1500 bytes. These are both valid assumptions for the realm of regular computations which we are dealing with.

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 elements of size S/p.
- CYCLIC : Elements of the array are distributed evenly across all the processors in a round robin fashion with each processor owning every pth element, 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.

Details of regular distributions can be found in [2, 1]. 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.

In considering costs for any type of array transfer from node i to node j, we have already seen that there will be three basic components : a sending component t^S_{ij}, a network component t^D_{ij}, and, a receiving component t^R_{ij}. We have also seen that t^S_{ij} is accounted for in the weight of node i, t^D_{ij} is taken to be the weight of the edge joining node i and node j, and, t^R_{ij} is accounted for in the weight of node j. The reason for doing this is that t^S and t^R require processor involvement, whereas t^D does not.

We propose the following expressions for the three cost components:

$$\begin{aligned} t^S_{ij} &= S_{ij}(p_i, p_j) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\ t^D_{ij} &= \frac{L}{p_i \cdot S_{ij}(p_i, p_j)} \cdot t_n \\ t^R_{ij} &= R_{ij}(p_i, p_j) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr} \end{aligned} \tag{13}$$

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^S_{ij}) 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 logic holds for the receiving component for 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.

It can be seen that the quantities S_{ij} and R_{ij} will depend on the kind of redistribution occurring. It is possible to express these quantities in terms of a pair of parameters of the sending and receiving

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

Table 1: Block Factors for Various Regular Distributions

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

Table 2: Skip Factors for Various Regular Distributions

distributions. The first of these parameters is called the Block Factor (BF), 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), it 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. We now write down the expressions for S_{ij} and R_{ij} as:

$$S_{ij} = \max(1, \frac{SF_j}{SF_i}, \frac{BF_i}{BF_j}, \frac{SF_j}{SF_i} \cdot \frac{BF_i}{BF_j})$$

$$R_{ij} = \max(1, \frac{SF_i}{SF_j}, \frac{BF_j}{BF_i}, \frac{SF_i}{SF_j} \cdot \frac{BF_j}{BF_i}) \tag{14}$$

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.

In all the expressions above, we have omitted some details in order to make them more understandable. First, we have considered only a one-dimensional array being transferred in all the cost functions. In practice, arbitrary n-dimensional arrays may be redistributed. In addition, the redistribution may not be confined to a single array, more than one array may need 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 have not shown these extended forms as they are complex and lengthy. Our actual implementation uses an extended form of these functions.

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 : A complete proof would require us to show that the statement above is true for all cases of redistributions. However, the lack of space prevents us from doing this, details can be found in [41]. Instead, we show that the statement holds for a pair of cases:

- Case A: BLOCK to BLOCK.

  For this case, the expressions for S_{ij} and R_{ij} (using Tables 1 and 2 and Equation 14) are given by:

$$S_{ij} = \max(1, \frac{p_j}{p_i})$$
$$R_{ij} = \max(1, \frac{p_i}{p_j}) \tag{15}$$

  Using these values in Equation 13, we obtain:

$$t_{ij}^S = \max(1, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps}$$
$$t_{ij}^D = \frac{L}{p_i \cdot \max(1, \frac{p_j}{p_i})} \cdot t_n \Rightarrow t_{ij}^D = \min(\frac{L}{p_i}, \frac{L}{p_j}) \cdot t_n \tag{16}$$
$$t_{ij}^R = \max(1, \frac{p_i}{p_j}) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr}$$

  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 1 and 2 and Equation 14) are given by:

$$S_{ij} = \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i})$$
$$R_{ij} = \max(1, \frac{p_i \cdot Y}{p_j \cdot X}, \frac{Y}{X}, \frac{p_i}{p_j}) \tag{17}$$

Using these values in Equation 13, we obtain:

$$t_{ij}^S = \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps}$$

$$t_{ij}^D = \frac{L}{p_i \cdot \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i})} \cdot t_n \Rightarrow t_{ij}^D = \min(\frac{L}{p_i}, \frac{L \cdot Y}{p_j \cdot X}, \frac{L \cdot Y}{p_i}, \frac{L}{p_j}) \cdot t_n \tag{18}$$

$$t_{ij}^R = \max(1, \frac{p_i \cdot Y}{p_j \cdot X}, \frac{Y}{X}, \frac{p_i}{p_j}) \cdot t_{ss} + L \cdot \frac{1}{p_j} \cdot t_{ps}$$

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 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. However, the lack of space prevents us from doing this, details can be found in [41]. Instead, we show that the statement holds for a pair of 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:

$$t_{ij}^S = \max(1, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps}$$

$$t_{ij}^R = \max(1, \frac{p_i}{p_j}) \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr} \tag{19}$$

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

$$t_{ij}^S \cdot p_i = \max(p_i, p_j) \cdot t_{ss} + L \cdot t_{ps}$$

$$t_{ij}^R \cdot p_j = \max(p_j, p_i) \cdot t_{sr} + L \cdot t_{pr} \tag{20}$$

  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:

$$t_{ij}^S = \max(1, \frac{p_j \cdot X}{p_i \cdot Y}, \frac{X}{Y}, \frac{p_j}{p_i}) \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps}$$
$$t_{ij}^R = \max(1, \frac{p_i \cdot Y}{p_j \cdot X}, \frac{Y}{X}, \frac{p_i}{p_j}) \cdot t_{ss} + L \cdot \frac{1}{p_j} \cdot t_{ps} \quad (21)$$

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

$$t_{ij}^S \cdot p_i = \max(p_i, \frac{p_j \cdot X}{Y}, \frac{X \cdot p_i}{Y}, p_j) \cdot t_{ss} + L \cdot t_{ps}$$
$$t_{ij}^R \cdot p_j = \max(p_j, \frac{p_i \cdot Y}{X}, \frac{Y \cdot p_j}{X}, p_i) \cdot t_{ss} + L \cdot t_{ps} \quad (22)$$

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

6 Optimality of the Allocation and Scheduling Method

While developing the Allocation algorithm, we assumed the existence of a perfect scheduling algorithm. Since the actual scheduling algorithm we use is not perfect, our methods may not achieve the optimum value in practice. The theoretical results that follow quantize the deviations of our algorithms from the best possible solution. In deriving these theorems, we have assumed that the underlying computation and communication cost functions are of the form discussed in the previous section.

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

Definition 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:

$$W_s = \sum_{i=1,n_b} t_{busy}^i \cdot p^i \quad (23)$$

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

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:*

$$T_{psa} \leq (1 + \frac{p}{p - PB + 1}) \cdot T_{opt}^{PB} \tag{24}$$

Proof:

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

$$W_{psa} \leq p \cdot T_{opt}^{PB} \tag{25}$$

If any node uses at most PB processors, we can say that the PSA being unable to schedule the next node immediately means it has at least $p - PB + 1$ processors busy currently. 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):

$$W_{psa} \geq (T_{psa} - \Delta) \cdot (p - PB + 1) + W_{\Delta} \tag{26}$$

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 greater than PB processors are idle, it means the PSA algorithm has a case when $PST < EST$ for all the unscheduled nodes (Refer Section 4). This implies that every other unexecuted node is dependent on the currently ongoing events which may be a node execution or a edge delay in progress. It is also clear that such a situation could occur many times in the building up of the 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 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 IS's 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:

$$\Delta \leq T_{opt}^{PB} \tag{27}$$

It can be seen that 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 27 in 26, we have:

$$W_{psa} \geq (T_{psa} - T_{opt}^{PB}) \cdot (p - PB + 1) \tag{28}$$

From Equations 25 and 28, we have:

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

$$\Rightarrow T_{psa} \leq (1 + \frac{p}{p - PB + 1}) \cdot T_{opt}^{PB} \tag{29}$$

which is the required result $\Box$.

Theorem 2 *In the first two steps of the PSA we modify the processor allocation produced by the convex programming formulation of Section 3. 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 the best possible scheduler, we have:*

$$T_{opt}^{PB} \leq (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot \Phi \tag{30}$$

where, Φ is the solution obtained from the convex programming formulation.

Proof: We first look at the effect of increasing or decreasing the number of processors used by the nodes of the MDG on the value of its average finish time and critical path time. This can be seen from the definition of these quantities in Section 3 and the cost functions of Section 5.

From this information, we can see that if we increase the allocation to any node i from p_i to p_i', its contribution to the average can increase by a factor of no more than $(\frac{p_i'}{p_i})^2$. This factor comes about because of the startup component in t_{ij}^R and t_{ij}^S. On the other hand, it is also evident that decreasing the processor allocation for any node will only decrease the value of the average.

Again, by looking closely at the material in the sections mentioned, we see that increasing the allocation to any node i from p_i to p_i' will increase the critical path by a factor no more than $\frac{p_i'}{p_i}$. This is because of the startup component in t_{ij}^R and t_{ij}^S. Similarly, decreasing the processor allocation of a node i from p_i to p_i' could also increase the critical path. This time the factor may be up to $(\frac{p_i}{p_i'})^2$. This is because of the structure of t_{ij}^D.

Having seen this, we now examine the effect of the initial steps of the PSA on the values of the average and critical path produced by the convex programming formulation (A_p and C_p).

In order to make our allocation practical, we first rounded-off the processor allocation in Step 1 of the PSA. Since we round-off to the nearest power of 2, it can be shown that the processor allocation for the ith node is changed at most by $\frac{1}{3}$ of its original value, i.e., p_i can decrease to $\frac{2 \cdot p_i}{3}$ or increase to $\frac{4 \cdot p_i}{3}$ in the worst case. Let the value of the average finish time 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 increase or decrease of processor allocation , we can write:

$$A_{RO} \leq (\frac{4}{3})^2 \cdot A_p \ ; \ C_{RO} \leq (\frac{3}{2})^2 \cdot C_p \tag{31}$$

After performing the round-off, we imposed a bound on the number of processors used by each node in Step 2. The value of PB we use is assumed to be a power of 2. If not, we would have to round-off again and might end up making some p_i's more that PB, which renders the bound useless. The net effect of this step is of a decrease in the processor allocation of some nodes, and no change in the processor allocation of others. The worst case decrease for any node is clearly from p to PB. If A_{PB} is the value of the average finish time and C_{PB} is the value of the critical path time for this bounded allocation, using the discussion on effects of processor increase or decrease, we have:

$$A_{PB} \leq A_{RO} \ ; \ C_{PB} \leq (\frac{p}{PB})^2 \cdot C_{RO} \tag{32}$$

Since T_{opt}^{PB} denotes the time obtained by using the best scheduler on this rounded-off and bounded processor allocation, we can write:

$$T_{opt}^{PB} = \max(A_{PB}, C_{PB}) \tag{33}$$

Using Equations 31 and 32 in the equation above we have:

$$T_{opt}^{PB} \leq \max((\frac{4}{3})^2 \cdot A_p, (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot C_p) \ \Rightarrow \ T_{opt}^{PB} \leq (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot \max(A_p, C_p) \tag{34}$$

From the equation above and the definition of Φ in Section 3, we have:

$$T_{opt}^{PB} \leq (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot \Phi \tag{35}$$

which is the required result □.

Intuitively, this theorem summarizes the effect of our rounding off and bounding steps. It tells us how much the solution can deviate from the optimal even if we used the best possible scheduler after having applied these steps. In the next theorem, we summarize all effects, i.e., using the PSA to schedule after the round-off and bounding steps.

Theorem 3 *Let T_{psa} denotes the value of the finish time obtained for a processor allocation using the convex programming formulation of Section 3 and the PSA. Then, we have:*

$$T_{psa} \leq (1 + \frac{p}{p - PB + 1}) \cdot (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \Phi \tag{36}$$

where, Φ is the solution obtained from the convex programming formulation.

Proof: This result is a direct consequence of the previous theorems (1 and 2) □.

Corollary 1 *The power of 2 that minimizes the value of the following expression is the optimum value of PB to use for the PSA:*

$$(1 + \frac{p}{p - PB + 1}) \cdot (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \tag{37}$$

Proof: From Theorem 3 it is clear that the expression to be minimized is the one given above. As we have discussed in Section 4, we must choose a PB that is a power of 2 or we may end up with an infeasible solution. A feasible solution is one in which the processor allocation for any node is a power of 2 as well as bounded by PB. Hence, the result □.

7 Implementation and Results

The allocation and scheduling algorithms proposed above were tried out on three benchmark MDGs. The MDGs were hand generated after studying the programs they correspond to and are shown in Figure 9. Our testbed machines were a 128 node Thinking Machines CM-5 and a 128 node Intel Paragon.

The first MDG corresponds to multiplication of two complex matrices of 128×128 elements. It has few nodes and is relatively simple. The other MDG we used corresponds to the Strassen's algorithm for multiplication of a pair of matrices of size 256×256 elements. This is a more complex MDG with many more nodes than the previous one. The book by Press et. al. [42] describes

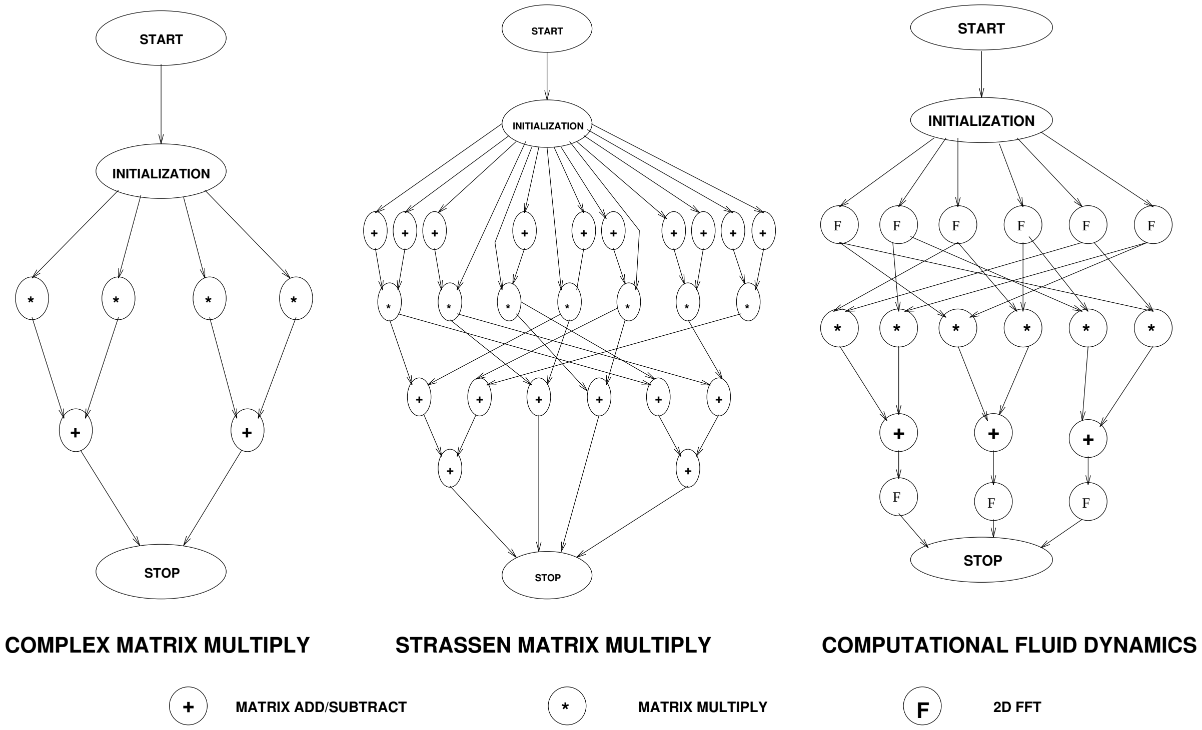


Figure 9: Benchmark MDGs Used

Strassen's algorithm in detail and explains its usefulness. Our third benchmark MDG corresponds to a Fourier-Chebyshev spectral Computational Fluid Dynamics (CFD) algorithm applied on a $128 \times 128 \times 65$ grid. Details of this algorithm can be obtained from [43]. The important routines used in our benchmark MDGs are Matrix Multiply, 2D FFT, Matrix Add, and, Matrix Subtract.

Having obtained the MDGs, we used MAST to study their execution profiles using 32, 64 and 128 processors on both target architectures. MAST generated the SPMD and the MPMD versions of code for all the benchmark MDGs so that we could compare the performance obtained for the two cases. For the SPMD case, every node in the MDG uses all the processors available; there are no data redistributions. For the MPMD case, we perform allocation and scheduling using our methods, data redistributions may be needed. The speedups and execution efficiencies obtained are shown in Figure 11 for the CM-5 and in Figure 12 for the Paragon. From these figures it can be seen that speedups obtained for the MPMD programs are much higher as compared to SPMD versions, especially, for larger systems. The only exception to this observation is the 32 processor case for the CFD algorithm on the CM-5. Here, the SPMD version performs slightly better than the MPMD version. This is because the data redistribution overhead for the MPMD program outweighs the gains obtained by efficient execution of the routines. In all other cases this

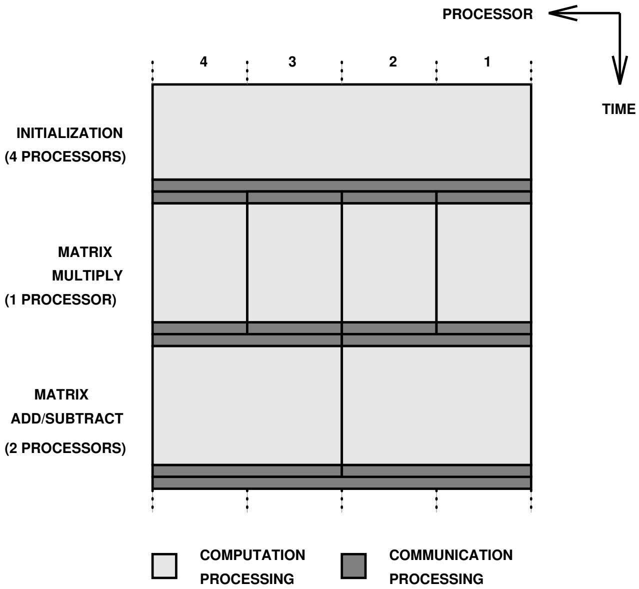


Figure 10: Allocation and Scheduling of Complex Matrix Multiply

Machine	Benchmark	Predicted Time (Secs)	Actual Time (Secs)	Error
CM-5	Complex Matrix Multiply	0.484	0.442	+9.5 %
	Strassen's Matrix Multiply	0.758	0.766	-1.0 %
	Computational Fluid Dynamics	0.467	0.426	+9.6 %
Paragon	Complex Matrix Multiply	0.161	0.187	-13.9 %
	Strassen's Matrix Multiply	0.288	0.306	-5.9 %
	Computational Fluid Dynamics	0.266	0.244	+9.0 %

Table 3: Predicted versus Actual Execution Times of Benchmark Programs for 64 Processors

overhead is more than amortized by the efficient execution of routines. The increased performance benefits obtained for larger systems makes allocation and scheduling critical for massively parallel computing. Intuitively, the benefits of using functional and data parallelism together will be greater when most of the available data parallelism in a routine has been exploited (this happens for large systems)

Another aspect of interest is the practicality of our models for processing and data transfer costs. In order to check this we have plotted the predicted and measured finish times of the three benchmark programs for a system size of 64 nodes for both target architectures in Table 3. The figure shows the close correspondence of the two quantities, which means our cost models are very practical.

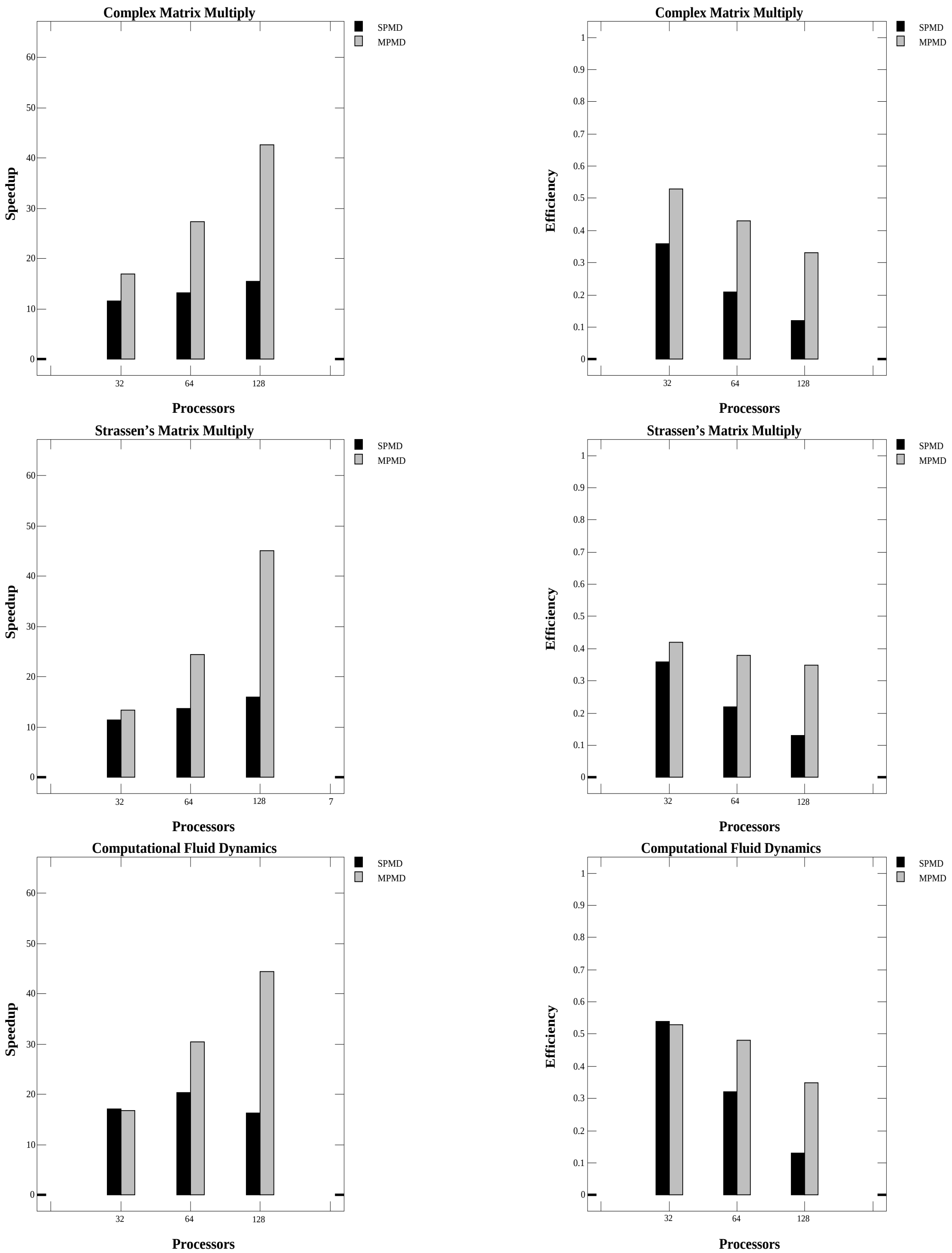


Figure 11: Speedup and Efficiency Comparison for SPMD and MPMD versions of Benchmark Programs on the Thinking Machines CM-5

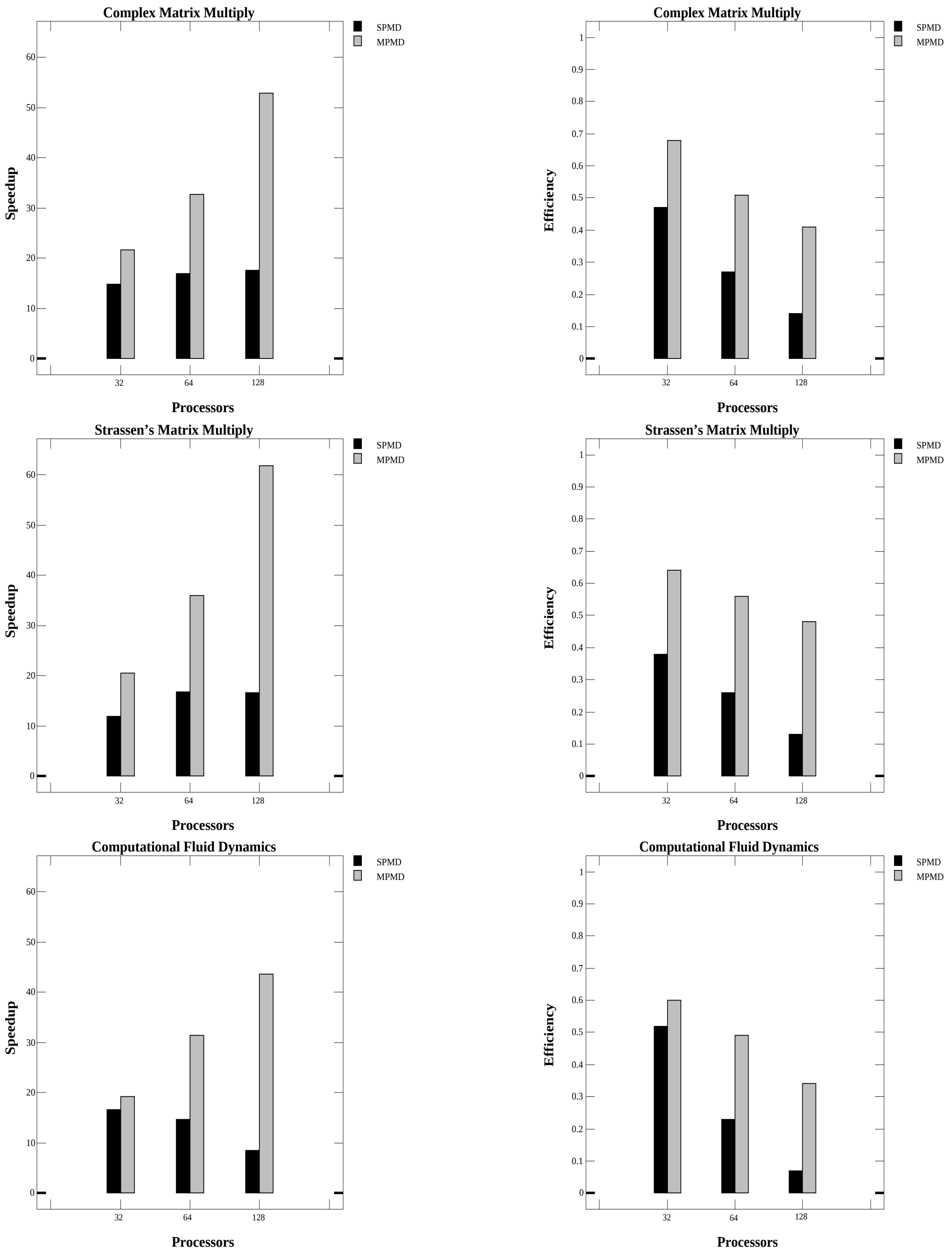


Figure 12: Speedup and Efficiency Comparison for SPMD and MPMD versions of Benchmark Programs on the Intel Paragon

8 Conclusions and Future Work

In this paper we have presented a framework for exploiting data and functional parallelism together. Basic to our framework is the MDG representation for a program. The MDG is constructed using a graphical programming tool. Costs for its nodes and edges are estimated using cost function models provided. We then use an allocation and scheduling approach on the MDG for exploiting functional and data parallelism together. Allocation is performed using a convex programming approach and scheduling is done using a variant of list scheduling. The performance of our allocation and scheduling approach has been theoretically analyzed; practical results have also been provided which show it is very effective.

In the future we will consider the following extensions:

- Using the SPMD compilation techniques developed for the PARADIGM compiler [9, 11, 10] to generate data parallel versions of the scientific library routines in MAST. Currently, we use hand coded parallel versions. Using the SPMD compilation will allow us to extend the scientific library in an easy manner.

- Development of static processing cost estimation techniques like the ones described in [8, 44]. This way, we can profile any user specified routine and not confine the user to using routines from the library provided in MAST.

- Minimization of redistribution costs by modifying the scheduling algorithm. Currently, this algorithm does not take data locality into account; by using such information, it may be able to avoid redistribution costs if the pair of nodes involved are executing on the same set of processors and have the same data distributions.

References

[1] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification*, 1993. Version 1.

[2] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling FORTRAN D for MIMD Distributed Memory Machines," *Communications of the ACM*, pp. 66–80, Aug. 1992.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim, "An Overview of a Compiler for Scalable Parallel Machines," in *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, (Portland, OR), August 1993.

[4] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K. Y. Wang, and M. Burke, "PTRAN II : A Compiler for High Performance FORTRAN," tech. rep., IBM T.J. Watson Research Center, 1993.

[5] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna FORTRAN," in *the Proceedings of the Third Workshop on Compilers for Parallel Computers*, pp. 145–164, 1992.

[6] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, "Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers, Design, Implementation, and Performance Results," *the Proceedings of the International Conference on Supercomputing*, pp. 351–360, July 1993.

[7] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Computing*, pp. 179–193, March 1992.

[8] M. Gupta, *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

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

[10] E. Su, D. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed Memory Multicomputers," *to appear in the Proceedings of the Parallel Architectures and Compiler Technology Conference*, August 1994.

[11] D. Palermo, E. Su, J. Chandy, and P. Banerjee, "Communication Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler," *to appear in the Proceedings of the International Conference on Parallel Processing*, August 1994.

[12] K. P. Belkhale and P. Banerjee, "Approximate Algorithms for the Partitionable Independent Task Scheduling Problem," in *the Proceedings of the International Conference on Parallel Processing*, (St. Charles, IL), pp. I:72–75, August 1990.

[13] K. P. Belkhale and P. Banerjee, "A Scheduling Algorithm for Parallelizable Dependent Tasks," in *the Proceedings of the International Parallel Processing Symposium*, (Los Angeles, CA), pp. 500–506, April 1991.

[14] S. Ramaswamy and P. Banerjee, "Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler," in *the Proceedings of the International Conference on Parallel Processing*, (St. Charles, IL), pp. II:134–138, August 1993.

[15] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism," *to appear in the Proceedings of the International Conference on Parallel Processing*, August 1994.

[16] A. Lain and P. Banerjee, "Techniques to Overlap Computation and Communication in Irregular Iterative Applications," *to appear in the Proceedings of the International Conference on Supercomputing*, July 1994.

[17] J. Subhlok, J. M. Stichnoth, D. R. 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), pp. 13–22, May 1993.

[18] I. T. Foster and K. M. Chandy, "FORTRAN M: A Language for Modular Parallel Programming," tech. rep., Argonne National Laboratory/California Institute of Technology, 1992.

[19] B. Chapman, P. Mehrotra, J. V. Rosendale, and H. Zima, "A Software Architecture for Multidisciplinary Applications: Integrating Task and Data Parallelism," Tech. Rep. TR 94-1, University of Vienna, 1994.

[20] A. L. Cheung and A. P. Reeves, "Function-Parallel Computation in a Data Parallel Environment," in *the Proceedings of the International Conference on Parallel Processing*, (St. Charles, IL), pp. II:21–24, August 1993.

[21] M. Girkar and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Transactions on Parallel and Distributed Computing*, pp. 166–178, March 1992.

[22] S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," tech. rep., University of Illinois, 1994.

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

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

[25] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, pp. 22–35, January 1978.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness.* Bell Laboratories, 1979.

[27] T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," in *the Proceedings of Supercomputing*, (Albuquerque, NM), pp. 633–642, November 1991.

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

[29] J. Subhlok, "Automatic Mapping of Task and Data Parallel Programs for Efficient Execution on Multicomputers," Tech. Rep. CMU-CS-94-106, Carnegie Mellon University, 1994.

[30] J. Subhlok, D. O'Halloran, T. Gross, P. A. Dinda, and J. Webb, "Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs," Tech. Rep. CMU-CS-94-106, Carnegie Mellon University, 1994.

[31] P. Dinda, T. Gross, D. O'Hallaron, E. Segall, J. M. Stichnoth, J. Subhlok, J. Webb, and B. Yang, "The CMU Task Parallel Program Suite," Tech. Rep. CMU-CS-94-131, Carnegie Mellon University, 1994.

[32] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, R. Wade, and V. Sunderam, "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing," in *the proceedings of the Scalable High Performance Computing Conference*, (Williamsburg, VA), pp. 129–136, April 1992.

[33] Message-Passing Interface Forum, *Document for a Standard Message-Passing Interface*, 1993. Version 1.0.

[34] I. Foster, M. Xu, B. Avalani, and A. Choudhary, "A Compilation System That Integrates High Performance FORTRAN and FORTRAN-M," in *the proceedings of the Scalable High Performance Computing Conference*, (Knoxville, TN), pp. 293–300, May 1994.

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

[36] J. Ecker, "Geometric Programming: Methods, Computations and Applications," *SIAM Review*, pp. 338–362, July 1980.

[37] C. L. Liu, *Elements of Discrete Mathematics.* McGraw-Hill Book Company, 1986.

[38] M. R. Garey, R. L. Graham, and D. S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Operations Research*, pp. 3–21, January 1978.

[39] Q. Wang and K. H. Cheng, "A Heuristic for Scheduling Parallel Tasks and Its Analysis," *SIAM Journal on Computing*, pp. 281–294, April 1992.

[40] J. W. Turek, J. and P. Yu, "Approximate Algorithms for Scheduling Parallelizable Tasks," in *the Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, 1992.

[41] S. Ramaswamy and P. Banerjee, "Modelling Data Redistribution Costs on Distributed Memory Multicomputers," tech. rep., University of Illinois, 1994.

[42] 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, 1988.

[43] 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, pp. 999–1028, 1991.

[44] T. Fahringer, *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, 1993.