# A Convex Programming Approach 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

Compilers have focussed on the exploitation of one of functional or data parallelism in the past. The PARADIGM compiler project at the University of Illinois is among the first to incorporate techniques for simultaneous exploitation of both. The work in this paper describes the techniques used in the PARADIGM compiler and analyzes the optimality of these techniques. It is the first of its kind to use realistic cost models and includes data transfer costs which all previous researchers have neglected. Preliminary results on the CM-5 show the efficacy of our methods and the significant advantages of using functional and data parallelism together for execution of real applications.

## 1. INTRODUCTION

Distributed memory multicomputers such as the Intel Paragon, the IBM SP-1 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.

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

- Generate data partitioning specifications [1, 2].
- Partition computations and generate communication for data parallel programs [3, 4, 5].
- Exploit functional and data parallelism [6].
- Provide compiler and runtime support for irregular applications [7].

*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.

For our discussion, we define *Functional Parallelism* to be any parallelism existent among the various loops(nest) in a given program and *Data Parallelism* to be parallelism within a loop (nest). To reiterate, these definitions are our own and may not correspond to popular versions.

### 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 Prasanna and Agarwal in [8]. The MDG is a weighted directed acyclic graph whose nodes correspond to loops (nest) of the given program and edges correspond to precedence constraints among these loops.

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 loop is called its processing cost. Processing costs will depend on the number of processors used to execute the loop and include all computation and communication costs incurred. For a loop's precedence constraints to be met, data transfer may be required between processors that execute it and the processors that process each of its predecessors. The time needed for data transfer between each such predecessor-successor pair of loops 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 loop, a network cost, and, a receiving cost for processors at the receiving loop. All of these cost components are a function of the number of processors used for the sending and receiving loops.

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 loop it corresponds to

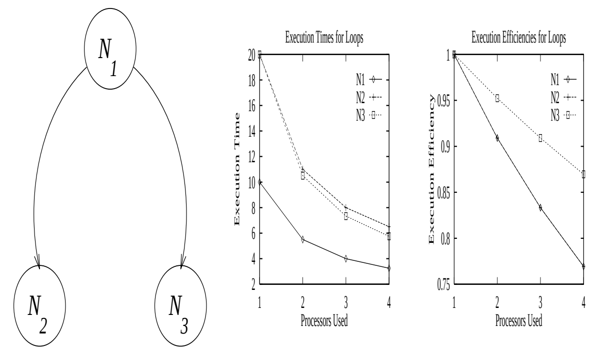


Figure 1: Example Showing Functional Parallelism

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

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 loops 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 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 loops 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 loops. 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 loops 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 loops use 4 processors. The second scheme on the other hand, is exploiting both functional and data parallelism, i.e., loops 2 and 3 execute concurrently as well as use 2 processors each.

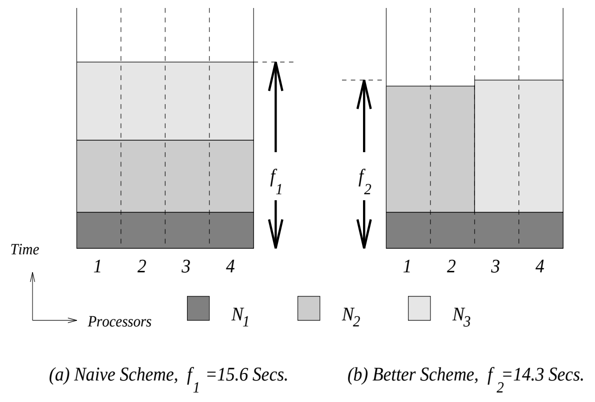

Time
Processors
f1
f2
N1
N2
N3
(a) Naive Scheme, f1 =15.6 Secs.
(b) Better Scheme, f2=14.3 Secs.

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 inefficiently as the size of the 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 is more efficient as compared to the first. This makes the second scheme execute the program faster than the first.

In summary, we can identify the issues involved in deciding on a good program execution strategy as:

1. Identification of the nodes and edges to be used in the MDG representation of the given program. We do not have any methods developed yet for this step. The possibility of using the work by Girkar and Polychronopoulos in [9] is being studied.
2. Determination of weights of nodes and edges of the MDG for the given target machine. We have proposed mathematical models for processing and data transfer costs in this paper but currently use actual measurements to determine the parameters of these models. This approach is similar to the Training Sets approach of Balasundaram et. al. in [10]. We are considering the use of static estimation techniques developed by Gupta and Banerjee in [2, 11] to try and eliminate the need for some of the measurements in the future.
3. Allocation of processors to nodes of the MDG and scheduling of these allocated nodes in a manner such as to minimize the execution time of the given program. This step has been the focus of work presented in this paper. We have not only developed algorithms to perform the allocation and scheduling, but also analyzed the quality of the results produced by these algorithms.
4. Generation of data parallel code for each node in the MDG based on the allocation produced by Step 3. Techniques already exist in the PARADIGM compiler for this step [3, 4, 5].
5. Using the generated code in Step 4 and the schedule produced in Step 3 to create an executable program

for each processor in the target system. It can be seen that the program created can be very different for each processor in the system. This type of programming is also called Multiple Program Multiple Data (MPMD) programming. This is in contrast to purely data parallel code where the program for each processor is similar to others in the system. Such type of programming is called Single Program Multiple Data (SPMD) programming.

### 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 [12]. Further treatment on this topic can also be found in the book by Garey and Johnson [13]. 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 [14], and Gerasoulis and Yang in [15, 16]. A bottom up approach considers the MDG to be made up of lightweight nodes (in terms of computation requirements) and coalesces these nodes together to form larger nodes during its construction of a schedule. The second is a top down approach like the ones used by Prasanna and Agarwal in [8], Belkhale and Banerjee in [17, 18], Ramaswamy and Banerjee in [6] 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 are 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 [8] 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 [17, 18] also does not consider the effects of non-zero data transfer costs. Their allocation and scheduling algorithms are similar to the ones we use. While data transfer costs were taken into account in the methods presented for allocation and scheduling in [6], we could not provide any bounds on the quality of our results due to the use of heuristics for allocation. In this paper we use exact methods (a convex programming formulation) for allocation and have developed theoretical bounds on the quality of our results. Another important difference between our previous work and this paper is that the cost models we use here for data transfer are considerably more accurate. A final difference is that the results presented here are based on real programs whereas our previous results were based on synthetic benchmarks.

In the next two sections we discuss our allocation and scheduling algorithms. We then present the processing and data transfer cost models used in Section 4. Theoretical results that discuss the optimality of our algorithms are in Section 5. Section 6 provides preliminary results obtained using our algorithms.

## 2. MDG ALLOCATION ALGORITHM

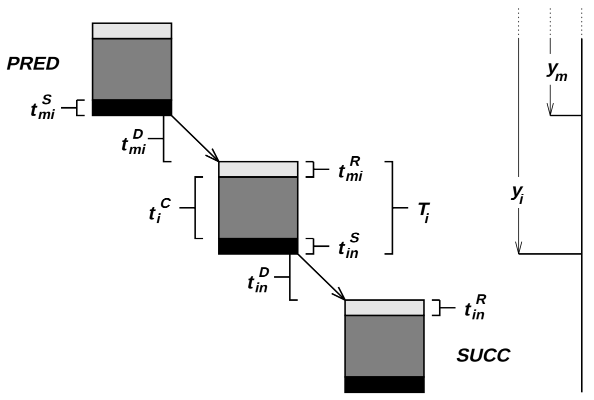
We first consider the problem of allocation of processors to the nodes of a given MDG. After the allocation is carried out using this algorithm, the MDG is ready to be scheduled using the algorithm described in the next section.

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 called START and node $n$ is called STOP. START precedes all nodes and STOP succeeds all nodes, either directly or indirectly. Intuitively, these correspond to the notion of FORK and JOIN nodes in the execution graph of a parallel program. These nodes could either be existing nodes or dummy nodes created for convenience.

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

minimize $\Phi$, where:

$$\begin{aligned}
&\Phi = \max(A_p, C_p)\\
&A_p = \tfrac{1}{p}\cdot\textstyle\sum_{i=1}^{n} T_i \cdot p_i\\
&C_p = y_n\\
&y_i = \max_{m\in PRED_i}(y_m + t^D_{mi}) + T_i\\
&T_i = (\textstyle\sum_{m\in PRED_i} t^R_{mi} + t^C_i + \sum_{n\in SUCC_i} t^S_{in})
\end{aligned}$$



where

1. $p_i$ represents the number of processors used by the $i$th node.
2. $t^C_i$ is the processing cost of the loop corresponding to node $i$ and is a function of $p_i$.
3. $t^R_{mi}$ represents the time required at node $i$ to process the messages it receives from predecessor node $m$ (receiving cost component of data transfer). $t^D_{mi}$ 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 $i$). $t^R_{in}$ 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 $i$th node.
7. $\Phi$ 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 thing is that $\Phi$ must be at least the 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 $\Phi$ must be at least the same as the critical path time.

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

Our formulation relies on the theory of *Convex Functions* described in [19] and the theory of *Posynomial Functions* described in [20]. We are unable to provide any details of these topics here due to space constraints. Basically, using the research on convex functions and posynomial functions, we can show that our formulation is equivalent to a *Convex Programming Formulation* if the following two 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. $t_{ij}^R \cdot p_j$, $t_{ij}^S \cdot p_i$ and $t_i^C \cdot p_i$ are also posynomial functions of the free variables.

Later, in Section 4, we present cost functions to represent the quantities $t_{ij}^D$, $t_{ij}^R$, $t_{ij}^S$, and $t_i^C$ that satisfy these conditions. We also demonstrate the accuracy 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 $\Phi$. 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 $\Phi$ time units. In practice, producing such a schedule is an NP-Complete problem [13]. Therefore, we use a scheduler as described in the next section which might produce a finish time different from $\Phi$. As we shall show in Section 5, we have quantified this deviation.

## 3. 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 two. This is done to make the final code generation very easy. The results we obtain in Section 6 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 $i$th 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 5. 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 recompute 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 all 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 [21], by Garey, Graham and Johnson in [22], by Wang and Cheng in [23],

| Node Name | $\alpha$ (%) | $\tau$ ($mS$) |
|---|---|---|
| Matrix Addition (64x64) | 6.7 | 3.73 |
| Matrix Multiply (64x64) | 12.1 | 298.47 |

Table 1: Parameters for Processing Cost Function

by Belkhale and Banerjee in [18], by Turek, Wolf and Yu in [24], and, by Ramaswamy and Banerjee in [6]. 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 5. 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.

## 4. 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, first, they have to be posynomial functions, and, second, they have to be accurate. Due to lack of space, we are unable to prove here that these functions are indeed posynomials. However, we establish the more important fact that they are indeed accurate.

The processing cost function we use is an often used model. The data transfer cost functions on the other hand are new and have been derived by us. Again, the lack of space prevents us from giving a detailed derivation of these functions. We have tried to give a flavor of the ideas behind the derivations.

**Processing Cost Model**

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

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

where $\tau_i$ is the execution time of the loop on a single processor and $\alpha_i$ is the fraction of the loop that has to be executed serially.

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

**Proof** : Omitted due to lack of space

It must be noted that the value of the parameter $\alpha_i$ need not necessarily be a constant. It may be a function of the number of processors used or the problem size. As long as it assumes a form that ensures both $t_i^C$ and $t_i^C \cdot p_i$ are posynomial functions with respect to $p_i$, our methods are applicable.

In order to evaluate the suitability of the form of our function, we performed the following experiment on the

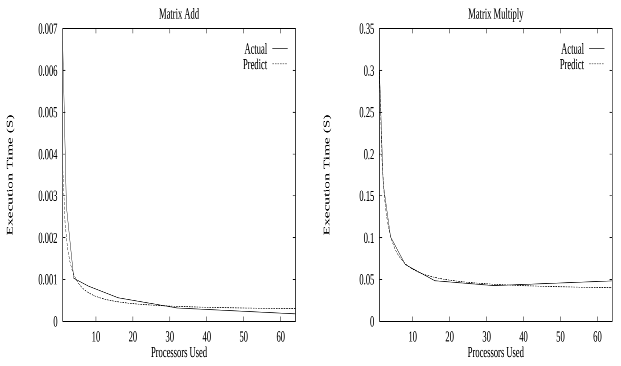


Figure 3: Actual versus Predicted Costs for Processing

$CM-5$. We first measured the cost of two of the loops used in our test programs, Matrix Multiply and Matrix Add, as a function of the number of processors used by them. We then used linear regression to determine the values of $\alpha$ and $\tau$ for each loop such that the costs predicted using our posynomial function fit the measured costs as best as possible. The results we obtained for the two loops are shown in Table 1. We have also shown the close fit of actual and predicted costs by plotting these together for the two loops in Figure 3. This figure clearly shows that the form we have chosen for the processing cost function is very suitable.

**Data Transfer Cost Model**

Here we consider the cost of transferring an array of data elements between 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. This is a valid assumption for the realm of regular computations that we are dealing with. The other assumption we make is that it is distributed along only one of its dimensions in a blocked manner. This assumption makes the model simple to construct and understand and does not necessarily limit its scope. Many real programs require only such distributions for good performance. For other programs more general distributions may be needed for optimal performance. Keeping this in mind, we are in the process of extending our cost functions.

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 $t_{ij}^S$, a network component $t_{ij}^D$, and, a receiving component $t_{ij}^R$. Again, we have seen that $t_{ij}^S$ is accounted for in the weight of node $i$, $t_{ij}^D$ is taken to be the weight of the edge joining node $i$ and node $j$, and, $t_{ij}^R$ 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.

For the programs we use as test cases, we deal with the transfer of 2 dimensional arrays. With the assumptions about distributions in mind, we can see that there are two possible ways in which such an array can be distributed; all processors sharing rows of the array equally or all pro-

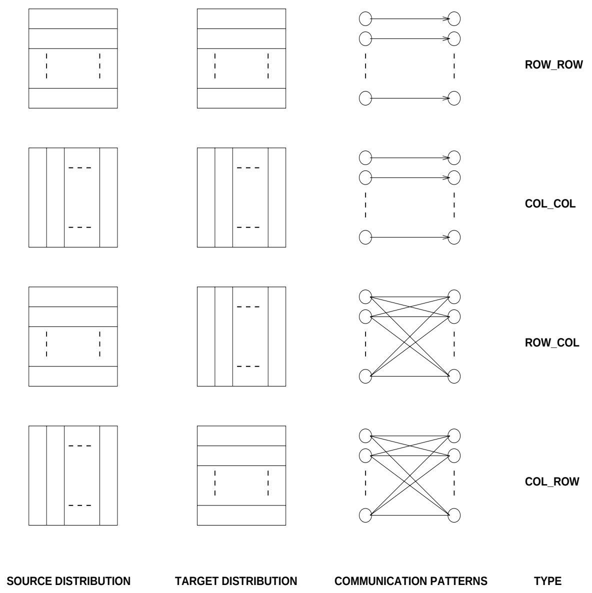


Figure 4: Inter Node Data Transfer Patterns

cessors sharing columns of the array equally. For example, if there were a $10 \times 10$ array to be distributed across 5 processors, we could either give them all 2 rows each or give them all 2 columns each. The types of data transfer that could occur between two nodes in the MDG for such arrays are illustrated in Figure 4. This figure assumes both nodes use the same number of processors for ease of illustration. In the general case, they could be different.

The first two cases in Figure 4 are called ROW2ROW and COL2COL and occur when the array is distributed along the same dimension in both sending and receiving processors. It can be seen that these cases are identical with respect to the time taken for transfer. We therefore refer to these cases jointly as the $1D$ type transfer. The cost components for the $1D$ case are given by:

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

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, $t_n$ is the network delay per message byte, and, $t_{sr}$, $t_{pr}$ are the startup and per byte cost for receiving messages. In deriving these costs, we have assumed that network costs are the same for all processor pairs. This assumption is valid for most of the current machines.

The other two cases in Figure 4 (ROW2COL and COL2ROW) occur when the array distribution is along distinct dimensions in the sending and receiving processors. Again, both cases are identical with respect to message transfer time. We refer to these cases jointly as the $2D$ type transfer. The cost components for this type are given by:

| $t_{ss}$ ($\mu Secs$) | $t_{ps}$ ($nSecs$) | $t_{sr}$ ($\mu Secs$) | $t_{pr}$ ($nSecs$) | $t_n$ ($nSecs$) |
|---|---|---|---|---|
| 777.56 | 486.98 | 465.58 | 426.25 | 0 |

Table 2: Parameters for Data Transfer Cost Functions

$$
\begin{aligned}
t_{ij}^{S} &= p_j \cdot t_{ss} + L \cdot \frac{1}{p_i} \cdot t_{ps} \\
t_{ij}^{D} &= L \cdot \frac{1}{p_i \cdot p_j} \cdot t_n \qquad (3) \\
t_{ij}^{S} &= p_i \cdot t_{sr} + L \cdot \frac{1}{p_j} \cdot t_{pr}
\end{aligned}
$$

where, the various parameters have the same meaning as mentioned before.

The differences in the $1D$ and $2D$ types of transfers arise due to differences in the number and size of the messages being sent between processors of the sending and receiving nodes. However, the net amount of data transferred for any given array has to be the same in both cases and is dependent on the size of the array.

In all the expressions above, we have omitted some details in order to make them more understandable. First, we have considered only one array being transferred in all the cost functions. In practice, this may not be true, i.e. multiple arrays may be transferred. Second, there may be both type of transfers occurring between a given pair of nodes in the MDGs, for example, one array may need a ROW2ROW type transfer while another may require a ROW2COL type transfer. It is easy to extend our functions to account for these effects. Our actual implementation uses an extended form of these functions.

**Lemma 2** *$t_{ij}^{S}$, $t_{ij}^{R}$ and $t_{ij}^{D}$ are all posynomial functions w.r.t. $p_i$ and $p_j$ for both $1D$, and, $2D$ cases.*

**Proof** : Omitted due to lack of space

In order to evaluate the form of our data transfer cost functions, we performed another experiment on the $CM-5$. First, we measured costs of transferring arrays using our routines between a varying number of sending and receiving processors. We then used linear regression to determine values of the parameters required in our functions such that the costs predicted using these functions fit the actual costs as best as possible. The results we obtained are shown in Table 2. We have also shown the close fit of actual and predicted costs by plotting these together for the two types of transfer in Figure 5. This figure again clearly shows the suitability of the form of our cost functions.

There is one point to note in Table 2, viz., $t_n$ being 0 for transfers. This is because of the way communication calls are implemented in the $CM-5$. In this machine, if a receive is called at a processor after the corresponding send has been completed, the data is actually transferred between processors at the time that the receive is called and not after the send is completed. The network cost per byte is therefore included in the processing cost per byte for a receive. While building the schedule for execution of the program using our methods, we always have receives

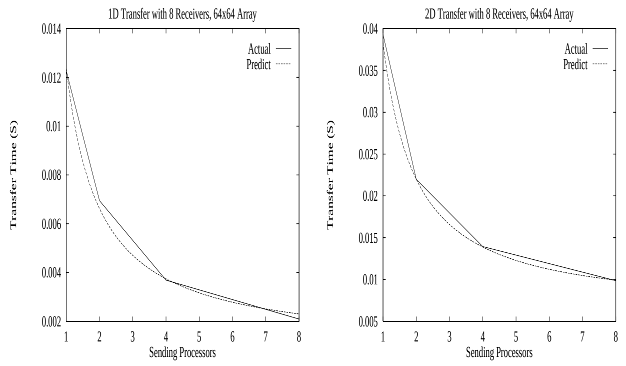


Figure 5: Actual versus Predicted Costs for Data Transfer

being called after the corresponding sends are completed. This occurs because predecessors of a node execute completely before the node is scheduled.

Finally, we must point out again that the approach we use for calculating parameters of the cost functions is similar to the one used by Balasundaram et. al. in [10]. The approach they use is called the Training Sets approach and involves running a set of programs on the target machine to perform measurements and then calculate the values of parameters used in their models.

## 5. OPTIMALITY ANALYSIS

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 quantify 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. As we have already shown in that section, this assumption is justified.

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:

$$W_s = \sum_{i=1,n_b} t^i_{busy} \cdot p^i \tag{4}$$

where, $t^i_{busy}$ is the $i$th 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 let $T^{PB}_{opt}$ denote the value obtained using the best possible scheduler. The relationship between these quantities is given by:*

$$T_{psa} \leq (1 + \frac{p}{p - PB + 1}) \cdot T^{PB}_{opt} \tag{5}$$

**Proof**:

In the best case the area of useful work done by the optimal scheduling algorithm can be $p \cdot T^{PB}_{opt}$. 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:

$$W_{psa} \leq p \cdot T^{PB}_{opt} \tag{6}$$

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 shall see later, this will not always be true. If the duration when this is not true is $\Delta$ (in the worst case), we can write (using the definition of useful work):

$$W_{psa} \geq (T_{psa} - \Delta) \cdot (p - PB + 1) + W_{\Delta} \tag{7}$$

Here we are assuming $W_{\Delta}$ 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 to Section 3). 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 $i$th such $IS$ ($IS_i$) control each of the the events of every subsequent $IS$ ($IS_j$ *for all* $j > i$). If this is 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 $T^{PB}_{opt}$ must be at least the length of the critical path, we can write:

$$\Delta \leq T^{PB}_{opt} \tag{8}$$

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 $W_{\Delta} \geq 0$. Using this inequality and Equation 8 in Equation 7, we have:

$$W_{psa} \geq (T_{psa} - T^{PB}_{opt}) \cdot (p - PB + 1) \tag{9}$$

From Equations 6 and 9, we have:

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

$$\Rightarrow T_{psa} \le (1 + \frac{p}{p - PB + 1}) \cdot T_{opt}^{PB} \qquad (10)$$

which is the required result □.

**Theorem 2** *In the first two steps of the PSA we modify the processor allocation produced by the convex programming formulation of Section 2. 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:*

$$T_{opt}^{PB} \le (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot \Phi \qquad (11)$$

*where, $\Phi$ 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 2 and the cost functions of Section 4.

From this information, we can see that if we increase the allocation to any node $i$ from $p_i$ to $p_i^{\prime}$, its contribution to the average can increase by a factor of no more than $(\frac{p_i^{\prime}}{p_i})^2$. 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 $p_i^{\prime}$ will increase the critical path by a factor no more than $\frac{p_i^{\prime}}{p_i}$. 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 $p_i^{\prime}$ could also increase the critical path. This time the factor may be up to $(\frac{p_i}{p_i^{\prime}})^2$. 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 $i$th 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:

$$A_{RO} \le (\frac{4}{3})^2 \cdot A_p$$

$$C_{RO} \le (\frac{3}{2})^2 \cdot C_p \qquad (12)$$

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:

$$A_{PB} \le A_{RO}$$

$$C_{PB} \le (\frac{p}{PB})^2 \cdot C_{RO} \qquad (13)$$

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

$$T_{opt}^{PB} = \max(A_{PB}, C_{PB}) \qquad (14)$$

Using Equations 12 and 13 in the equation above we have:

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

$$\Rightarrow T_{opt}^{PB} \le (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot \max(A_p, C_p) \qquad (15)$$

From the equation above and the definition of $\Phi$ in Section 2, we have:

$$T_{opt}^{PB} \le (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \cdot \Phi \qquad (16)$$

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 2 and the PSA. Then, we have:*

$$T_{psa} \le (1 + \frac{p}{p - PB + 1}) \cdot (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \Phi \qquad (17)$$

*where, $\Phi$ 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:*

$$(1 + \frac{p}{p - PB + 1}) \cdot (\frac{3}{2})^2 \cdot (\frac{p}{PB})^2 \qquad (18)$$

**Proof**: From Theorem 3 it is clear that the expression to be minimized is the one given above.

As we have discussed in Section 3, 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 both bounded by $PB$ as well as a power of 2.

Hence, the result □.

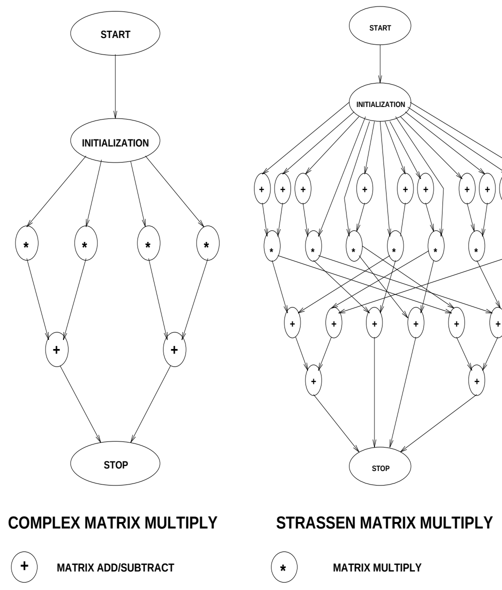


Figure 6: Benchmark MDGs Used

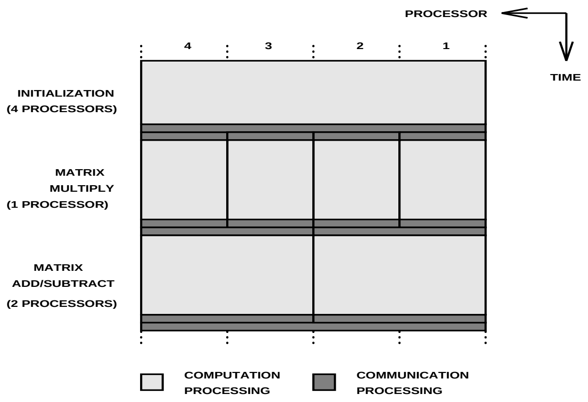

PROCESSOR
TIME
4
3
2
1
INITIALIZATION (4 PROCESSORS)
MATRIX MULTIPLY (1 PROCESSOR)
MATRIX ADD/SUBTRACT (2 PROCESSORS)
COMPUTATION PROCESSING
COMMUNICATION PROCESSING

Figure 7: Allocation and Scheduling for Complex Matrix Multiply

## 6. IMPLEMENTATION AND RESULTS

The allocation and scheduling algorithms proposed above were tried out on two MDGs. The aim of this exercise was to try and see the effectiveness of these algorithms as compared to naive schemes for these graphs. The MDGs were hand generated after studying the programs they correspond to and are shown in Figure 6. Our testbed machine was a 64 Node Thinking Machines CM-5.

The first MDG corresponds to multiplication of two complex matrices. It has few nodes and is relatively simple. The other MDG we used corresponds to the Strassen's algorithm for matrix multiply. This is a more complex MDG with many more nodes than the previous one. The book by Press et. al. [25] describes Strassen's algorithm in detail and explains its usefulness. There are three basic types of loops for both MDGs, viz., Matrix Initialization, Matrix Multiplication and Matrix Addition. All the data transfers are of the $1D$ type in both algorithms. As we have seen in Section 4, all the parameters for the cost functions

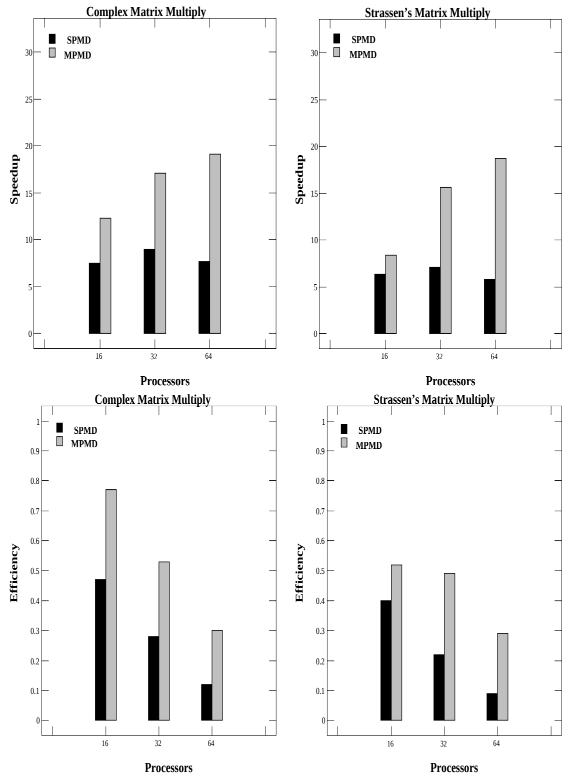


Figure 8: Speedup and Efficiency Comparison for SPMD and MPMD versions of Test Programs

corresponding to the routines and the transfers have been obtained for the CM-5.

Having obtained the MDGs, we used our allocation and scheduling algorithms to obtain the best allocation and scheduling scheme to be used for system sizes of 16, 32 and 64 processors. In Figure 7 we show an example of the allocation and schedule obtained for Complex Matrix Multiply on a 4 processor system. Using these allocation and scheduling schemes, we hand generated an MPMD program for the target machine (64 node CM-5). The SPMD versions corresponding to these MPMD programs (All nodes in the MDG use all $p$ processors) were also hand coded. The MPMD and SPMD versions were executed and timed. The speedups and execution efficiencies obtained are shown in Figure 8. From this figure it can be seen that speedups obtained for the MPMD programs are much higher as compared to SPMD versions, especially for larger systems.

Another aspect of interest is the accuracy 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 two test programs for different system sizes in Figure 9. The figure shows that the two quantities are fairly close to each other, which means our cost models are fairly accurate in practice.

Finally, recall Section 5, where we proved theoretical bounds on the deviation of the quantity $T_{psa}$ from the quantity $\Phi$. We wanted to check what the deviation was in practice. For this, we compared the values of the quantities

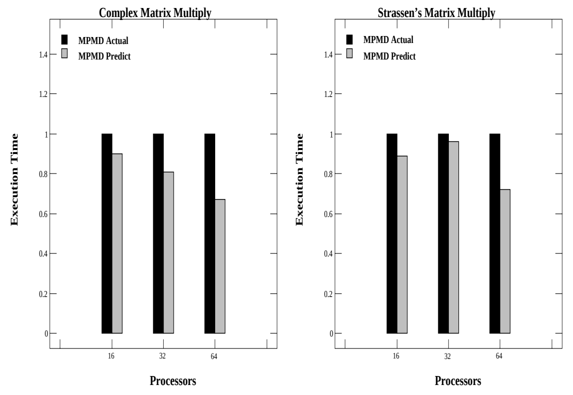


Figure 9: Predicted versus Actual Execution Times of Test Programs (Normalized to Actual Times)

| Program Name | System Size | $\Phi$ (S) | $T_{psa}$ (S) | Percent Change |
|---|---|---|---|---|
| Complex Matrix Multiply ($64 \times 64$) | 16 | 0.117 | 0.114 | -2.6 |
| | 32 | 0.075 | 0.074 | -1.3 |
| | 64 | 0.054 | 0.055 | -1.9 |
| Strassen's Matrix Multiply ($128 \times 128$) | 16 | 0.125 | 0.136 | +8.8 |
| | 32 | 0.222 | 0.236 | +6.3 |
| | 64 | 0.077 | 0.085 | +15.6 |

Table 3: Deviation of $T_{psa}$ from $\Phi$ for Test Programs

$T_{psa}$ and $\Phi$ as produced by our allocation and scheduler for the two test programs for various system sizes. This is presented in Table 3. The table shows that the deviation is very small in practice, i.e. we are able to achieve near optimal solutions using our methods.

## REFERENCES

[1] 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.

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

[3] 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*, pp. II:30–38, 1993.

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

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

[6] S. Ramaswamy and P. Banerjee, "Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler," in *Proceedings of the International Conference on Parallel Processing*, 1993.

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

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

[9] 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.

[10] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer, "A Static Performance Estimator to Guide Data Partitioning Decisions," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1991.

[11] M. Gupta and P. Banerjee, "Compile-time Estimation of Communication Costs on Multicomputers," in *International Parallel Processing Symposium*, 1992.

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

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

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

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

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

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

[18] K. P. Belkhale and P. Banerjee, "A Scheduling Algorithm for Parallelizable Dependent Tasks," in *International Parallel Processing Symposium*, pp. 500–506, 1991.

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

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

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

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

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

[24] J. W. Turek, J. and P. Yu, "Approximate Algorithms for Scheduling Parallelizable Tasks," in *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pp. 323–332, 1992.

[25] 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.