Parallel Processing Letters


# SIMULTANEOUS ALLOCATION AND SCHEDULING USING CONVEX PROGRAMMING TECHNIQUES *

Shankar Ramaswamy and Prithviraj Banerjee
*(shankar,banerjee@crhc.uiuc.edu)*
*Computational Science and Engineering,*
*469 Computer Systems and Research Lab,*
*University of Illinois,*
*1308 W. Main St.*
*Urbana, IL 61801*

Received February 1995
Revised August 1995
Communicated by (Gerasoulis and Yang)

ABSTRACT

Simultaneous exploitation of task and data parallelism provides significant benefits for many applications. The basic approach for exploiting task and data parallelism is to use a task graph representation (Macro Dataflow Graph) for programs to decide on the degree of data parallelism to be used for each task (allocation) and an execution order for the tasks (scheduling). Previously, we presented a two step approach for allocation and scheduling by considering the two steps to be independent of each other. In this paper, we present a new simultaneous approach which uses constraints to model the scheduler during allocation. The new simultaneous approach provides significant benefits over our earlier approach for the benchmark task graphs that we have considered.

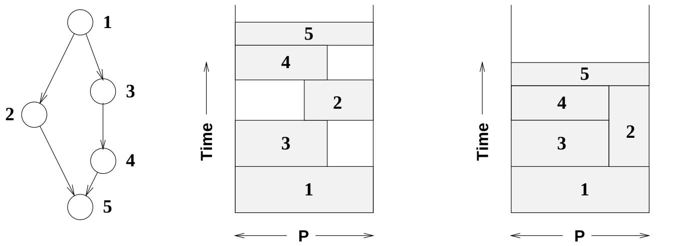
*Keywords:* Allocation, Scheduling, Convex Programming, Task and Data Parallelism

## 1. Introduction

The research outlined in this paper is motivated by the benefits obtained through simultaneous exploitation of task and data parallelism[1,2,3] in massively parallel distributed memory multicomputers such as the Intel Paragon, IBM SP-1/SP-2 and the Thinking Machines CM-5. Exploiting task and data parallelism involves using a task graph representation for programs to decide on the number of processors to use for each task (allocation) and an execution scheme for the allocated tasks (schedule). Previously[1,4], we had outlined a two step approach for allocation and scheduling which performed a reasonable job for our benchmark application task graphs. In that approach we assumed the existence of a "perfect scheduler" while

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

Fig. 1. Example to motivate constraint based allocation




(a) Task graph (b) Unconstrained Allocation (c) Constrained Allocation

doing allocation and did not account for the characteristics of a practical scheduling algorithm like our list scheduling variant. In this paper, we discuss an approach for constrained allocation that accounts for the characteristics of the scheduler. The usefulness of such an approach can be considered using a simple task graph such as the one shown in Figure 1(a). Parts (b) and (c) of the figure show schedules for two possible allocations for the task graph. The first allocation does not taken into account the possible scheduling conflict that can occur between task 2 and tasks 3 or 4. If we account for the scheduling conflict using constraints on the allocation, we obtain an allocation that has a shorter schedule length.

The problem of optimally scheduling a set of tasks with precedence constraints on a $P$ processor system when each task uses just one processor has been shown to be NP-complete by Lenstra and Kan[5]. Our allocation and scheduling problem is considerably harder than this basic scheduling problem. Approximate solutions to the allocation and scheduling problem are based on either a top-down method or a bottom-up method. Top-down methods assume the MDG consists of heavyweight tasks and break them down. Bottom-up methods assume the MDG consists of lightweight tasks and cluster them. We use a top-down approach in this paper. Other top-down approaches include the work by Prasanna and Agarwal[6] and by Belkhale and Banerjee[7]. The principal differences between the work by these researchers and ours are that our cost models are far more realistic and that we consider edge costs that are ignored by all the others. Some of the research on bottom-up approaches includes work by Sarkar[8] and by Yang and Gerasoulis[9].

The next two sections contain necessary background material. This is followed by a description of our previous two step allocation and scheduling approach and the new simultaneous approach. We then present results in support of our new approach and conclude.

## 2. Convex and Posynomial Functions

In this section we provide a brief overview of the theory of convex and posynomial functions. Details can be found in Luenberger's book[10] and the paper by Ecker[11].

**Definition 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 $\alpha$, $0 \leq \alpha \leq 1$, the point $\alpha\mathbf{x}_1 + (1-\alpha)\mathbf{x}_2 \in C$.*

**Definition 2 Convex Functions**
*A function $f$ defined on a convex set $\Omega$ is said to be* convex *if, for every $\mathbf{x}_1$, $\mathbf{x}_2 \in \Omega$, and every $\alpha$, $0 \leq \alpha \leq 1$,*

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

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

**Definition 3 The Convex Programming Problem**
*The convex programming problem is stated as follows:*

$$\textit{minimize } f(\mathbf{x}) \textit{ such that } \mathbf{x} \in S \tag{2}$$

*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 making hill-climbing out of local minima unnecessary.

**Definition 4 Posynomial Functions**
*A* posynomial *is a function $g$ of a positive variable $\mathbf{x} \in \mathbf{R}^{\mathrm{n}}$ that has the form*

$$g(\mathbf{x}) = \sum_j \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \tag{3}$$

*where the exponents $\alpha_{ij} \in \mathbf{R}$ and the coefficients $\gamma_j > 0$.*

Posynomials are similar to polynomials, except that the coefficients $\gamma_j$'s must be positive and the exponents $\alpha_{ij}$'s could be any real number unlike in polynomials where they need to be integers. 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 useful property of posynomials is that they can be mapped onto convex functions through an elementary variable transformation[11] $x_i = e^{z_i}$. Such a transformation maps the problem of minimizing a posynomial function under posynomial constraints to a convex programming problem.

If $f$ and $g$ are convex functions defined on a convex set $S$, then:

- **Sum Property** The functions $f + g$ is a convex function over $S$.
- **Constant Property** The function $c \cdot f$, $c \geq 0$ 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 described before, posynomials can be transformed to convex functions using a simple variable transformation. 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. Macro Dataflow Graphs

The task graph representation we use for programs is called the Macro Dataflow Graph (MDG). An example MDG is shown in Figure 1(a). The MDG is a weighted directed acyclic graph whose nodes represent data parallel tasks and edges represent precedence. Currently, we consider precedences that are due to data dependences; control dependences are not accounted for. There are two distinguished nodes in the MDG – START and STOP. START precedes all other nodes and STOP succeeds all other nodes. The nodes of an $n$ node MDG are numbered 1 through $n$; START is numbered 1 and STOP is numbered $n$.

The weights of nodes and edges in the MDG are based on the concepts of Processing and Data transfer costs. Processing costs account for the communication and computation costs of tasks and depend on the number of processors used to execute the tasks. Data transfer costs account for communication that is necessary for preserving data dependence relationships between tasks; these costs depend on the number of processors used by the pair of dependent tasks. Data transfer costs have three components - a Sending cost for processors of the preceding task, a Network cost that accounts for delays in the network before messages reach the processors of the succeeding task and a Receiving cost for processors of the succeeding task.

The cost of a node is the sum of the receiving cost components for all incoming data transfers, the processing cost of the task it corresponds to and the sending cost components for all outgoing data transfers. The cost of an edge is the network cost component of the corresponding data transfer. We have assumed mathematical models for these costs as a function of the number of processors used for each of the tasks in the MDG. These models are briefly described below; details are in[4].

### *3.1. Processing and Data Transfer Cost Models*

We use Amdahl's law for the processing cost. If the routine corresponding to the $i$th node uses $p_i$ processors, its execution time ($t_i^C$) is given by:

$$t_i^C = (\alpha_i + \frac{1-\alpha_i}{p_i}) \cdot \tau_i \tag{4}$$

where $\tau_i$ is the execution time of the routine on a single processor and $\alpha_i$ is the serial fraction of the routine.

For the data transfer cost, we make the following assumptions for a transfer of $L$ bytes between a pair of tasks $(i, j)$, where $i$ is the preceding task and $j$ is the succeeding task:

- The $L$ bytes are evenly spread across the $p_i$ sending processors before the transfer and across the $p_j$ receiving processors after the transfer.
- Each sending processor sends $S_{ij}$ messages. Each receiving processor receives $R_{ij}$ messages. $S_{ij}$ and $R_{ij}$ are functions of $p_i$ and $p_j$. We have provided methods for the calculation of these quantities in[4].

Based on these assumptions, we use the following cost functions for the sending ($t_{ij}^S$), network ($t_{ij}^N$) and receiving ($t_{ij}^R$) components of a data Transfer between tasks $i$ and $j$:

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

$$t_{ij}^N = \frac{L}{p_i \cdot S_{ij}(p_i, p_j)} \cdot t_n \tag{5}$$

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

where, $t_{ss}$ and $t_{ps}$ are the startup and per byte cost for sending messages from a processor, $t_n$ is the network cost per message byte, and, $t_{sr}$ and $t_{pr}$ are the startup and per byte cost for receiving messages at a processor.

The cost models described above are accurate in practice and can all be shown to be posynomial functions; details are provided in[4].

### 3.2. Node and Edge Costs

We can write down the cost of the $i$th node as:

$$T_i = \sum_{p \in PRED_i} t_{pi}^R + t_i^C + \sum_{s \in SUCC_i} t_{is}^S \tag{6}$$

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

The cost of the edge between tasks $i$ and $j$ is given by:

$$E_{ij} = t_{ij}^N \tag{7}$$

Using the properties of posynomials described in Section 2, we can show the node and edge costs to be posynomials.

### 3.3. MDG Properties

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

$$C = y_n$$

$$y_i = \max_{p \in PRED_i} (y_p + E_p i) + T_i \tag{8}$$

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

$$A = \frac{1}{P} \cdot \sum_{1}^{n} T_i \cdot p_i \tag{9}$$

The critical path represents the longest path in the MDG and the average area provides a measure of the processor-time area required by the MDG. Again, using the properties of posynomials, we can show the critical path and average area to be posynomials. As we shall see in the next section, these properties are useful while making allocation and scheduling decisions.

## 4. Two Step Allocation and Scheduling

Our previously reported two step approach for allocation and scheduling (2STEP) given an MDG and a $P$ processor system is[1,4]:

1 Allocate processors to nodes in the MDG using the Convex Programming Allocation Algorithm.
2 Schedule the allocated nodes using the Prioritized Scheduling Algorithm.

### *4.1. Convex Programming Allocation Algorithm (CPAA)*

The CPAA for a given MDG and a given $P$ processor system has the following steps:

1 Obtain a real number solution to the following minimization problem:

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

$$\textit{subject to } 1 \leq p_i \leq P \; \forall \; i = 1, n \tag{10}$$

2 Round-off the set of $p_i$'s obtained in the previous step to integers. This set of $p_i$'s is the allocation for the given MDG.

The intuition behind minimizing $\Phi$ is that it represents a theoretical lower bound on the finish time that can be obtained for a given MDG. This is because the finish time can neither be smaller than the longest path in the graph nor produce a processor-time product that is less than that produced by the average area. By producing an allocation that concentrates on minimizing this lower bound, we are hoping to produce the lowest possible finish time. However, this minimization process is carried out in the continuous domain which results in $p_i$'s which are real numbers rather than integers; we assume that rounding-off the real number solution will produce an allocation close enough to one that can be produced by using an integer minimization heuristic. Our reasons for using a continuous domain formulation are:

- The posynomial properties of our cost models make our minimization problem equivalent to a convex programming formulation[4]. This guarantees our finding the minimum solution.
- A convex programming formulation has been shown to be solvable in polynomial time[12,13] – this means we can find a minimum allocation in polynomial time.

Note that in the CPAA, our search space for a set of $p_i$'s is not constrained in any form, tasks are allowed to use even the complete system if needed. This unconstrained minimization does not ensure "schedulability" - by this term we mean that allowing a task to use any number of processors may result in an inability to schedule tasks that can run concurrently with it. As we shall see in the next section, our simultaneous allocation and scheduling approach modifies the CPAA to make allocations more “schedulable”.

### *4.2. Prioritized Scheduling Algorithm (PSA)*

The steps involved in the PSA are:

1 Place the START node of the MDG in a queue called the Ready queue and mark its Earliest Start Time ($EST_1$) as 0.
2 Pick a node $i$ from the Ready queue that has the lowest value of the Earliest Start Time ($EST_i$). Use the schedule built thus far to check the time at which the processor requirement of the task can be met. This is called the Processor Satisfaction Time ($PST_i$) for the node. Schedule the node at $\max(EST_i, PST_i)$ and compute its finish time ($FT_i$).
3 If the node scheduled in the previous step is STOP, terminate the scheduler with a finish time $FT_n$; else, proceed to the next step.
4 Check all the successors of the node just scheduled to see if all the precedence constraints of any of them have been met, i.e. all their predecessors have been scheduled. Any successor nodes that meet this criterion are placed on the Ready queue with their Earliest Start Time computed.
5 Repeat starting at Step 2.

Note that picking the node with lowest Earliest Start Time in Step 2 of the algorithm creates a priority among nodes, hence the name for the algorithm. The scheduler described above is a variant of the popular list scheduling algorithm which has been used by numerous researchers including Liu[14], Garey, Graham and Johnson[15], and Wang and Cheng[16]. Some of these researchers have also used variants of the basic list scheduling algorithm. In the case where the maximum number of processors used by any node of the MDG is bounded, we can show the PSA to produce a schedule within a factor of the optimum[1,4].

The complexity of the CPAA step has been shown to be $O(n^{2.5})$[12,13] and the complexity of the PSA is $O(np\log p)$. The $p\log p$ term arises in Step 2 of the PSA where the processor finish times need to be sorted in order that a suitable set of processors be determined for the next task. Therefore, the complexity of the 2STEP approach is $O(n^{2.5} + np\log p)$.

## 5. Simultaneous Allocation and Scheduling

Our new simultaneous allocation and scheduling approach (SIMUL) given an MDG and a $P$ processor system is:

1 Allocate $P$ processors to all the nodes in the MDG and schedule using the Modified Prioritized Scheduling Algorithm.
2 Note down the finish time of the schedule produced.
3 Repeat the following steps $X$ times
4 Allocate processors to nodes in the MDG using the Constrained Convex Programming Allocation Algorithm.

5 Schedule the allocated nodes using the Modified Prioritized Scheduling Algorithm.
6 Check finish time of schedule to see if it betters the best finish time; if it does, store the allocation found and update the best finish time.

### *5.1. Constrained Convex Programming Allocation Algorithm (CCPAA)*

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

1 Obtain a real number solution to the following minimization problem:

$$\begin{aligned} &\text{minimize } \Phi = \max(A, C) \\ &\textit{subject to } 1 \leq p_i \leq P \ \forall\ i = 1, n \\ &\textit{and constraint set } CS \ : \sum_{i \in CURR_j} p_i \leq P \ \forall\ j = 1, NCON \end{aligned} \tag{11}$$

The description of the MPSA algorithm that follows will clarify the concept of the constraint set.
2 Round-off the set of $p_i$'s obtained in the previous step to integers. This set of $p_i$'s is the allocation for the given MDG.

### *5.2. Modified Prioritized Scheduling Algorithm (MPSA)*

The steps involved in the MPSA are:

1 Place the START node of the MDG in a queue called the Ready queue and mark its Earliest Start Time ($EST_1$) as 0. Initialize the the Number of Constraints ($NCON$) to 0 and the Constraint Set ($CS$) to $\{\emptyset\}$.
2 Pick a node $i$ from the Ready queue that has the lowest value of the Earliest Start Time ($EST_i$). Use the schedule built thus far to check the time at which the processor requirement of the task can be met. This is called the Processor Satisfaction Time ($PST_i$) for the node. Schedule the node at $\max(EST_i, PST_i)$ and compute its finish time ($FT_i$).
3 If, in the previous step, the node $i$ is scheduled at $PST_i$, i.e. $PST_i > EST_i$, increment $NCON$ and add $CURR_{NCON}$ to the constraint set $CS$, where $CURR_{NCON}$ is given by:

$$CURR_{NCON} = \{j \mid FT_j \geq PST_i\} \tag{12}$$

Note that node $i$ is included in $CURR_{NCON}$ and that nodes considered for inclusion are only those that have already been scheduled. [a]
4 If the node scheduled in the previous step is STOP, terminate the scheduler with a finish time $FT_n$; else, proceed to the next step.

[a] The $CURR$ set for our example of Figure1(b) would be $2, 3, 4$.

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

6 Repeat starting at Step 2.

### *5.3. Comments*

The basis for our simultaneous allocation and scheduling approach is to model scheduling conflicts in the allocation process, thereby making the allocation more "schedulable". As the scheduling conflicts depend on the allocation, we iterate through the allocation and scheduling steps a few times to get a good solution. Instead of iterating for a fixed number of times, we could monitor the finish time of our schedule and stop when it begins to increase. However, during our experiments we found that temporary increases in the schedule time can sometimes occur. The reason for this is that the constraint set takes a few iterations to stabilize. The number of times we iterate ($X$) is a parameter for our allocation and scheduling algorithm. In practice, we find a value of $X = 10$ is sufficient to find a good solution. Larger values of $X$ do not provide much improvement in the quality of solutions.

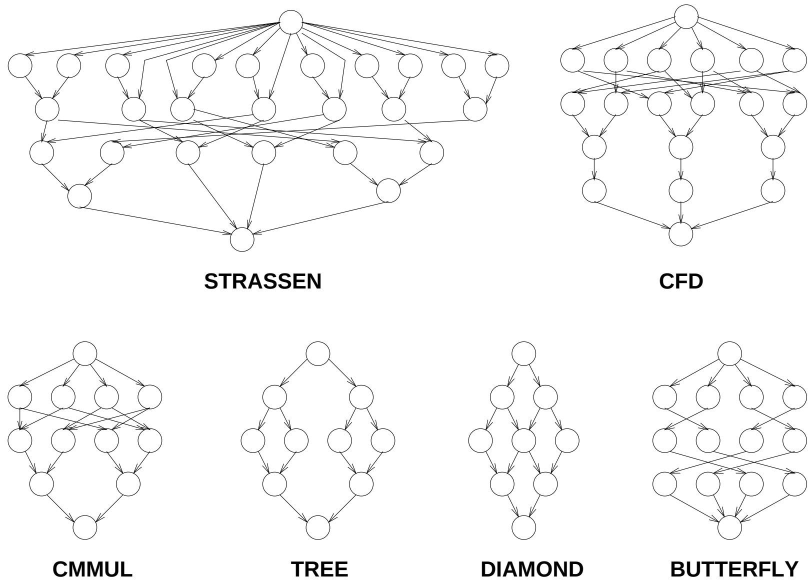
The constraints we apply to our minimization problem can be easily shown to be posynomial functions. This leads to our preserving the fact that our minimization is equivalent to a convex programming formulation, thus providing us with all the benefits outlined in the previous section. As our results in the next section will show, our new simultaneous allocation and scheduling approach provides improves the finish times obtained for benchmark MDGs appreciably and is very robust.

The addition of constraints does not change the complexity of solving a convex programming formulation, therefore the complexity of the CCPAA step remains $O(n^{2.5})$[12,13]. Again, the the complexity of the MPSA remains $O(np\log p)$ since Step 2 is still the dominating step in the algorithm. Therefore, the complexity of the SIMUL approach is $O(X(n^{2.5} + np\log p))$. As we have discussed before, a value of 10 for $X$ is suitable for good quality results, which makes the complexity of the SIMUL approach $O(n^{2.5} + np\log p)$ for all practical purposes.

## 6. Results

In order to test our simultaneous allocation and scheduling approach, we used a set of eight benchmark MDGs. The structure of a few of these MDGs is shown in Figure 2. Three of our benchmark correspond to real programs and have been used in the past. STRASSEN is the multiplication of a pair of matrices using Strassen's algorithm, CFD is a spectral computational fluid dynamics code and CMMUL is a complex matrix multiply benchmark. The cost model parameters for the tasks in these MDGs were determined using actual measurements on the Thinking Machines CM-5. We have five synthetic MDGs in our list – BUTTERFLY, DIAMOND, TREE, RANDOM1 and RANDOM2. The cost model parameters for tasks in these MDGs were generated randomly. The reason for using synthetic MDGs was to

Fig. 2. Some benchmark MDGs.




ensure the robustness of our approach.

We performed allocation and scheduling on the benchmark MDGs using our two step approach (2STEP) and our simultaneous approach (SIMUL). We also used two other schemes for comparison; the first allocates one processor to each task in the MDG (ONE) and the other allocates the full system for each task (ALL). ONE corresponds to a pure task parallel execution of the MDG and ALL corresponds to a pure data parallel execution of the MDG.

We used three system sizes for our experiments - 32, 64 and 128 processors. We have reported the speedups obtained using each scheme in Table 3; the last two columns also provide a ratio of the speedup obtained using the SIMUL scheme to those obtained using the ALL and 2STEP approaches. From the table, we can make the following observations:

- The ALL approach performs better that the ONE approach for all our benchmarks, the reason is that there is more data parallelism than task parallelism in the benchmarks.
- Both our allocation and scheduling approaches - 2STEP and SIMUL perform much better than ONE or ALL. This makes a strong case for using a mix of task and data parallelism.
- SIMUL outperforms 2STEP in most cases; in others it does as well as 2STEP. Therefore, our approach of using constraints on allocation to model scheduling is worthwhile.

## 7. Conclusions and Future Work

We have presented a new simultaneous approach for allocation and scheduling of Macro Dataflow Graphs. Our work is motivated by the benefits of simultaneous

Table 1. Speedups obtained using various schemes.

| Benchmark | $P$ | Scheme | | | | $\frac{SIMUL}{ALL}$ | $\frac{SIMUL}{2STEP}$ |
|---|---|---|---|---|---|---|---|
| | | ONE | ALL | 2STEP | SIMUL | | |
| STRASSEN | 32 | 6.9 | 12.4 | 20.3 | 20.3 | 1.64 | 1.00 |
| | 64 | 6.9 | 16.4 | 33.9 | 33.9 | 2.06 | 1.00 |
| | 128 | 6.9 | 18.8 | 52.9 | 57.0 | 3.04 | 1.08 |
| CFD | 32 | 4.5 | 18.7 | 14.9 | 23.7 | 1.27 | 1.59 |
| | 64 | 4.5 | 22.7 | 28.4 | 42.3 | 1.86 | 1.49 |
| | 128 | 4.5 | 18.9 | 46.1 | 68.0 | 3.60 | 1.48 |
| CMMUL | 32 | 3.9 | 11.2 | 19.0 | 19.0 | 1.69 | 1.00 |
| | 64 | 3.9 | 14.9 | 28.7 | 29.8 | 2.00 | 1.04 |
| | 128 | 3.9 | 17.0 | 41.6 | 42.7 | 2.52 | 1.03 |
| BUTTERFLY | 32 | 4.7 | 11.8 | 14.3 | 21.3 | 1.81 | 1.49 |
| | 64 | 4.7 | 14.3 | 22.8 | 35.0 | 2.45 | 1.54 |
| | 128 | 4.7 | 15.9 | 35.8 | 50.1 | 3.14 | 1.40 |
| DIAMOND | 32 | 1.9 | 11.2 | 12.0 | 15.1 | 1.35 | 1.26 |
| | 64 | 1.9 | 13.5 | 15.1 | 19.0 | 1.41 | 1.25 |
| | 128 | 1.9 | 14.9 | 17.5 | 21.6 | 1.45 | 1.24 |
| TREE | 32 | 4.5 | 12.3 | 14.6 | 19.3 | 1.57 | 1.32 |
| | 64 | 4.5 | 15.0 | 26.7 | 33.3 | 2.21 | 1.25 |
| | 128 | 4.5 | 16.9 | 36.9 | 51.2 | 3.03 | 1.39 |
| RANDOM1 | 32 | 2.9 | 12.4 | 15.2 | 17.4 | 1.40 | 1.14 |
| | 64 | 2.9 | 15.1 | 21.3 | 26.6 | 1.76 | 1.25 |
| | 128 | 2.9 | 17.0 | 26.4 | 31.8 | 1.87 | 1.21 |
| RANDOM2 | 32 | 2.3 | 11.5 | 15.7 | 17.7 | 1.53 | 1.13 |
| | 64 | 2.3 | 13.9 | 21.1 | 24.1 | 1.74 | 1.14 |
| | 128 | 2.3 | 15.5 | 25.1 | 28.8 | 1.86 | 1.15 |

exploitation of task and data parallelism. The basis for our simultaneous allocation and scheduling approach is to use constraints to model the scheduling process while performing allocation using a convex programming formulation. This is in contrast to our earlier two step approach of performing unconstrained allocation first and then scheduling the allocated nodes. As our results show, the simultaneous approach provides significant performance benefits over the two step approach.

In the future, we are planning to investigate if any other types of constraints can be added besides the type suggested here to provide further performance benefits. We shall also consider the allocation and scheduling problem for MDGs which have edges due to control dependence.

## References


1. S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism," in *the Proceedings of the International Conference on Parallel Processing*, (St. Charles, IL), pp. II:116–125, August 1994.

2. 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.

3. I. T. Foster and K. M. Chandy, "FORTRAN M: A Language for Modular Parallel Programming," *Journal of Parallel and Distributed Computing*, vol. 26, pp. 24–35, April 1995.

4. S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," Tech. Rep. UILU-ENG-94-2213/CRHC-94-10, University of Illinois, 1994.

5. J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, vol. 26, pp. 22–35, January 1978.

6. 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.

7. 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.

8. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.

9. 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.

10. D. G. Luenberger, *Linear and Nonlinear Programming*. Addison-Wesley, 1984.

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

12. P. M. Vaidya, "A new algorithm for minimizing convex functions over convex sets," in *Proceedings of the Symposium on Foundations of Computer Science*, (Research Triangle Park, NC), pp. 332–337, October 1989.

13. P. M. V. Sachin S Sapatnekar, Vasant B Rao and S.-M. Kang, "An Exact Solution to the Transistor Sizing Problem for CMOS Circuits Using Convex Optimization," *IEEE Transactions on Computer-Aided Design*, vol. 12, pp. 1621–1634, November 1993.

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

15. M. R. Garey, R. L. Graham, and D. S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Operations Research*, vol. 26, pp. 3–21, January 1978.

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