STATIC ALLOCATION FOR A DATA FLOW MULTIPROCESSOR SYSTEM

T.M. Ravi
M. D. Ercegovac
T. Lang
R. R. Muntz

November 1986
CSD-860028

Static Allocation For A Data Flow Multiprocessor System

T. M. Ravi, M. D. Ercegovac, T. Lang and R. R. Muntz
Computer Science Department, University of California, Los Angeles, California 90024

This paper describes an algorithm for compile-time allocation of any acyclic data flow graph to a general multiprocessor system. The single-pass algorithm incorporates a variable communication time associated with a task which depends on whether successor tasks are allocated to the same or different processors. The algorithm is based on two principles :- precedence to critical tasks and minimization of communication time. The performance of the algorithm is studied and compared favorably with critical path algorithms.

For effective allocation of fine grain data flow graphs with large control and communication overhead, the graph is preprocessed by a graph reduction algorithm. The graph reduction algorithm forces the graph to be of appropriate size prior to allocation. Performance studies indicate that graph reduction of fine grain graphs leads to significant decrease in response time.

1. INTRODUCTION

The data flow model of computation naturally supports concurrent processing by distributing sequencing and control of the program. Programs are represented as data flow graphs with nodes representing computation and arcs representing data dependencies. Sequencing is done by the flow of data in an asynchronous manner (data-driven sequencing). A node is activated when all its inputs arguments have arrived. Moreover, as the control is decentralized many nodes can be activated simultaneously.

Data flow principles can be applied at any level where the exploitation of concurrency would lead to a cost-effective increase in performance. Sequencing at the task level (task-flow) and at the machine instruction level are two examples with different degrees of granularity. Of particular importance in developing cost-effective architectures based on the data flow approach, is the choice of the granularity of nodes which depends on the ratio of communication and processing time and the power of the dependency analysis of the user program. The size of partitions and hence the granularity of tasks is determined to a large extent by the ease with which the user or the user's tools can determine dependencies in the program.

Task allocation is the assignment of tasks to processors in order to optimize a performance measure, given the system characteristics (e.g., number of processors and communication delays). Two main approaches exist for task allocation. In *static* allocation, the tasks are allocated at compile time to processors using global information about the program and the system organization. The cost of allocating tasks is incurred once for a given program even though it may be executed repeatedly. However, static allocation policies can be inefficient when estimates of run-time dependent characteristics are inaccurate.

A *dynamic* allocation policy is based on measuring processor loads at run-time and assigning activated tasks to the least loaded processor. Dynamic policies use limited information about the program behavior, but can balance the load at run-time by migrating tasks. The disadvantage of dynamic allocation is the overhead to measure the load of processors, calculation of global minimum load (or at least the minimum over a neighborhood of processors), and allocation of tasks at run-time. In addition global optimizations based on the critical path in the program are not possible.

In our work we adopt a static allocation policy which is suitable for our real-time application, where a single program is run repeatedly on a multiprocessor system. We describe an algorithm for compile time allocation of data flow graphs to a general multiprocessor system. The algorithm is tuned for an existing multiprocessors system described in Section 2.1.

2. THE MODEL

2.1 Architecture

The target architecture is the SANDAC IV multiprocessor [4], consisting of up to 16 Motorola MC68000 based processors connected by a global bus. It has been modified [5] for data flow execution of task graphs. The system is designed for real-time applications with periodic gathering of data from external sensors and processing it to drive output actuators.

Communication between processors is message based [13]. Results from a task running on a processor can be sent to a successor task on the same processor or on a different processor. Communication within a processor, termed *local communication*, is dominated by the time taken to activate tasks. Since data flow execution tends to have significant control overhead due to waiting-matching, task-fetch units and token labelling [2]. Communication to a task on a different processor, termed *bus communication*, is more time consuming, as the global bus also has to be accessed. Because of the characteristics of the communication mechanism [13], the following constraints result:

- The execution of tasks and the communication of results cannot be overlapped as the processor is busy during communication.
- Results are sent out only when the task has completed execution.
- If a task has several results, each result has to be sent out sequentially.

The *processing time* of a task is calculated at compile-time from the total number of cycles taken by the instructions in the task and the instruction cycle time. The local communication time can similarly be calculated by analyzing the mechanisms for task initialization in the target data flow architecture. The bus communication is estimated from the amount of data to be communicated, the bus time and the delay in accessing the communication network. Since the bus communication time depends on the load, this time is roughly estimated by considering the average delays caused by a medium load on the communication network.

2.2 Computation

We consider the allocation of a program at compile-time (*static allocation*) to a system with n identical processors. The program is partitioned at compile-time into tasks of different size. Partitioning compilers [6] transform programs into data flow graphs with nodes representing tasks and arcs representing precedence relationships between tasks. The partial ordering of tasks necessary for correct execution is captured by the dependencies between these tasks. There is no restriction on the granularity of the nodes of the graph, and each

task may contain any number of instructions to be executed sequentially.

The nodes of the graph have a single point of entry and a single point of exit, i.e., a task is activated when all its input (arguments) have arrived, and can deliver its results to successor tasks only after execution is complete. A single entry point implies that each task is a complete self-contained portion of the computation, and is not made to wait for intermediate inputs once its execution has begun. Having a single exit point is a restriction imposed by our target architecture.

Tasks are the basic unit of allocation and are indivisible over processors. Preemption of tasks is not allowed and a task once started will run until completion.

We assume the data flow graph to be acyclic without conditional nodes. It is assumed to have a single entry node and a single exit node. As usual, any graph can be converted to a single entry single exit graph by introducing a dummy entry and exit node.

We consider a deterministic model, where processing and communication times of tasks are known a priori. Associated with each task we have :

$t_p{}^i$ - processing time of task T_i

$t_{cl}{}^{ij}$ - local communication time for results between task T_i and task T_j

$t_{cb}{}^{ij}$ - bus communication time for results between task T_i and task T_j

The processing time (t_p) of a task is an estimate of the time taken to execute the task in a processor. The communication time (t_c) of a task has components due to both local and bus communication. When results are sent to successor processes which reside on the same processor then a lower local communication time (t_{cl}) is incurred. Bus communication time (t_{cb}) is incurred if the results have to be sent to a task on a different processor. The results from a tasks are sent out sequentially and hence the total communication time (t_c) of a task is the sum of the communication times of individual results. The communication time ($t_c{}^i$) for task T_i, is the time taken to communicate all the results to the successor tasks and is given by

$$t_c^i = \sum_{Task\ T_j \in\ same\ processor} t_{cl}^{ij} + \sum_{Task\ T_j \in\ different\ processor} t_{cb}^{ij} \quad (1)$$

During the execution of task T_i the processor is busy for $(t_p^i + t_c^i)$ time units.

3. TASK ALLOCATION

3.1 Introduction

We now describe a static task allocation algorithm that reduces the *response time* of the program on a system with n processors. Since optimal allocation of a graph with precedences is known to be *NP-complete* [16], we develop an heuristic algorithm. Our algorithm is based on *list schedules*, in particular *critical-path list schedules*, which have been reported in [1, 15]. We include the effect of communication times associated with each task, by modifying the critical-path list schedule reported by Kohler [15].

A *critical-path list scheduling algorithm* consists of a list of tasks ordered according to the longest path from the task to the exit node of the graph (critical path); and assigns to an idle processor the first task (i.e. the one with largest critical path) from the list that can be executed (i.e. whose arguments are available). The application of this algorithm to the case that includes communication times as defined in the previous section, encounters three specific problems:

1. Since the communication time associated with a task can be local or bus communication time depending on where its successors are executed, to reduce the overall response time it is convenient to allocate a task and its successors to the same processor. Consequently, the allocation algorithm has to include some mechanism for this.

2. The communication time associated with a particular task is not known before allocation, since it depends on the processors on which the successors will be executed. Consequently, it is not possible to calculate the precise critical paths.

3. The execution time for a particular task (processing + communication) depends on the

allocation of the successors. Because of this it is not possible to precisely estimate when the processor will be free again.

These problems are taken care of in our algorithm as follows. To reduce the communication time, instead of choosing the first task on the list that can be executed, we select from a set of *candidates* (as described in the algorithm of the next sub-section) the one that produces the maximum saving in communication time.

It turns out that our algorithm does not rely on an exact calculation of the critical paths. This is because instead of choosing the top task in the list ordered according to critical paths, we choose the top several candidates, whose critical path falls within a certain range. Consequently, the effect of the different ordering of tasks produced due to the lack of precision in determining the critical paths, is minimized by selecting a task for allocation from the top few tasks on the list instead of the top task on the list. Therefore, in our algorithm for an estimate of the critical paths, we include with each task just the local communication times of the results.

Finally, to be able to calculate the execution time of each task once it has been allocated, we reverse the graph and perform the allocation on the reversed graph. To obtain the actual schedule it is sufficient to reverse the schedule obtained. This procedure solves the problem because now a task is allocated after its actual successors, and therefore its communication times are known. To have the correct task charged with the communication time in the reversed graph, the communication times are associated with input arcs. As illustrated in Figure 1, the cost of communication t_c^{35} is incurred by the processor which executes task T_3, but cannot be estimated when task T_3 is allocated because the task T_5 has not yet been allocated. However if we reverse the graph and then apply the allocation algorithm to the graph, then T_5 is allocated before T_3. In the reversed graph, the cost of communication t_c^{35} is associated with the input arc to T_3, and charged to the task T_3. Now as task T_5 is allocated before T_3, hence t_c^{35} can be estimated when T_3 is allocated. Consequently, reversing the graph before allocation allows us to maintain correct bookkeeping when the value of communication time depends on the allocation of successor nodes.

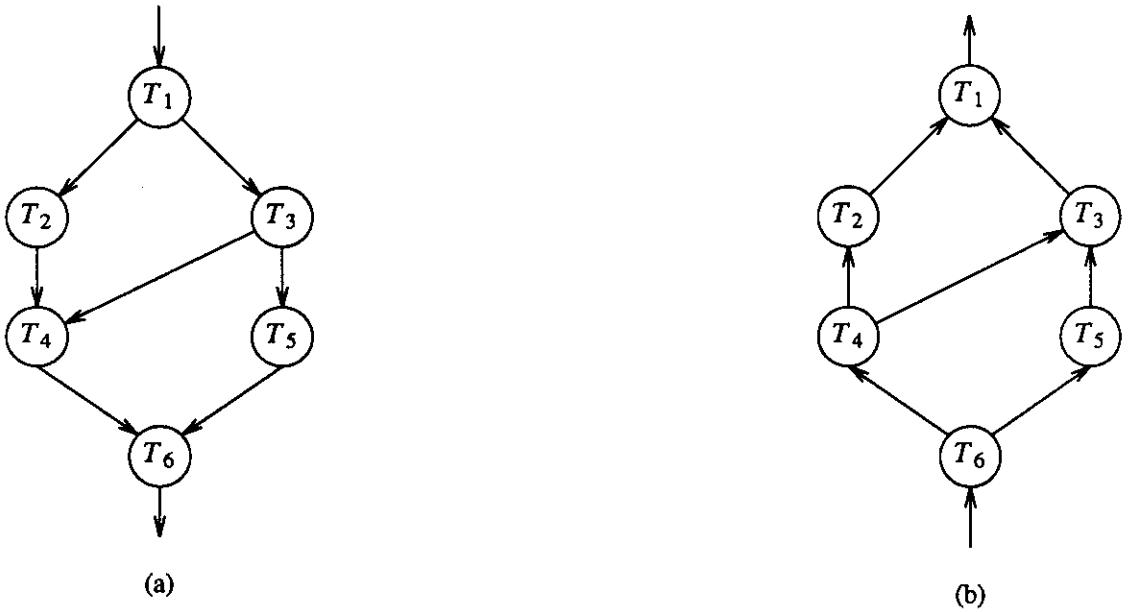

Figure 1 : Graph Reversal

Hence as the input to the allocation algorithm, we provide the reversed version of the original data flow graph with communication times associated with the input arcs.

3.2 The Algorithm

Consider a graph with tasks $T_1, T_2 \ldots T_k$, to be executed on n processors $P_1, P_2 \ldots P_n$. The arcs of the graph represent precedences in the graph. Two lists are constructed - a *Processor list* (L_p) and *Task list* (L_t).

The processor list L_p, contains the processors listed in increasing order of busy times. The processor on top of the list is the one which will become free next. Initially, the processors are in any order in the list, as they are all free.

The task list L_t consists of unallocated tasks in decreasing order of their critical path lengths. The critical path length CP(T_i) of a task T_i, is defined to be the length of the longest path from the exit node to T_i. In the calculation of critical paths, we take the lower value of the communication time (i.e., the local communication time) for the arcs. The critical paths of nodes in a graph can be evaluated recursively, starting at

the exit node T_k. The task list L_t is initially generated by sorting the tasks in decreasing order of their critical paths. When a task has been allocated it is removed from the task list L_t.

The allocation algorithm is as follows :

1. Choose the top processor P_a from the processor list L_p, which is the next processor to become idle.

2. Scan the task list L_t from the top, until the first candidate for execution in processor P_a is found. A task is a candidate for execution if it has been activated, i.e., if all its immediate predecessors have completed execution and delivered their results.

3. Continue to scan the task list and choose as candidates all activated tasks with critical path within a deviation Δ from the critical path of the first candidate. Stop scanning the task list as soon as a task which does not fall within the deviation Δ is encountered.

4. Select amongst the candidates the task T_{SEL} to be assigned to processor P_a. The task which, when allocated to the processor P_a, gives the maximum saving in communication time is selected. The saving in communication time is the sum of the difference of the bus and local communication time for each immediate predecessor task assigned to the same processor P_a.

5. The busy time of the processor P_a is updated. The processor will be busy for the period $(t_p^i+t_c^i)$ from the time it starts executing the task (T_{SEL}). The communication time (t_c^i) is calculated from Eq. (1).

6. The processor P_a is reinserted at the appropriate position in the processor list L_p, which is ordered according to increasing busy time.

7. The task T_{SEL} is removed from the task list L_t.

8. If no activated tasks are found in L_t for allocation to P_a, i.e. if no candidates are found in Step (2), then processor list L_p is scanned for the first processor P_b whose busy time is greater than the busy time of P_a. This amounts to waiting for a processor to complete execution of a task, and checking again if any

new tasks can be activated.

Processor P_b is removed from its position and placed on the top of the processor list L_p. Processor P_a and any other processors with busy time equal to that of P_a are updated with busy time equal to the busy time of P_b, thus creating idle times in processors when no tasks are ready.

9. Go back to Step (1) and continue this procedure until the task list (L_t) is empty.

The allocation algorithm is described in detail in Ravi [19]. The output of the algorithm is a list of tasks which have been allocated to each processor. The list can be reversed to obtain a schedule for execution.

3.3 Graph Reduction

When the processing times of tasks are small compared to the communication times then *thrashing* can occur because of excessive communication overhead and the allocation algorithm is not as effective as the graph has not been properly partitioned. This is because at the time of allocation of a task there is a limited view of the surrounding subgraph and little freedom to force the allocation of the subgraph to which the task belongs to a single processor. Consequently, when the original data flow graph is fine grain we can frequently achieve better response times from allocation by preprocessing the graph and transforming it to a larger grain task graph. Graph reduction is an additional heuristic to force tasks, which are the unit of allocation, to be of appropriate size. Replacing subgraphs by single nodes thereby reducing the fine grain graph into a variable resolution task graph, forces the allocation of subgraphs to a processor when the parallelism within the subgraph cannot be effectively utilized. Moreover the time taken for the allocation process is reduced as less nodes have to be allocated now. Other work on the exploitation of larger grain data flow parallelism include Babb [3], Ercegovac [7, 8], Gajski [9, 10], Gaudiot [11, 12], Hwang [14], and Ravi [20].

The appropriate size of tasks depends on a tradeoff between the efficiency of the algorithm to exploit parallelism and the overhead due to communication and activation. The criterion used here for lumping together nodes into a single task is the local minimization of the response times for subgraphs under consideration.

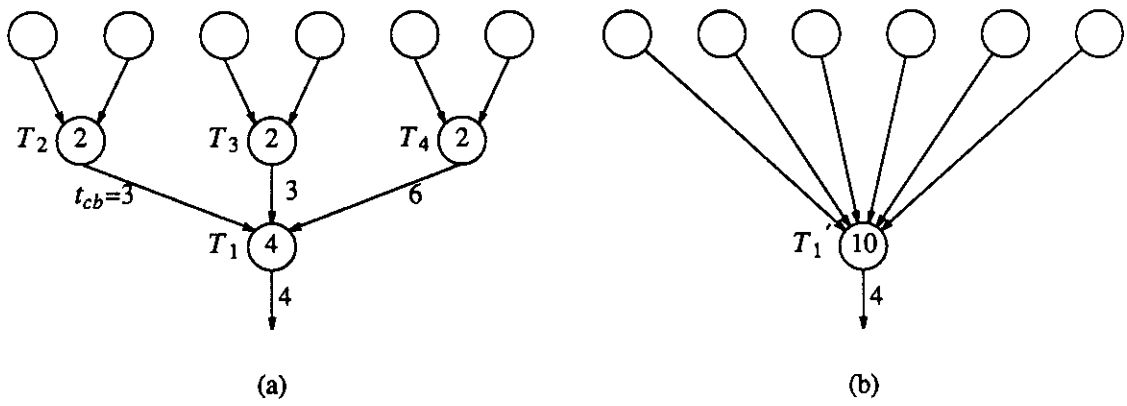

Figure 2 : Graph Reduction

Consequently, the condition for combining a node T_1 with its predecessor nodes, provided each of the predecessor nodes has a single result, is:

$$\sum_i t_p^{T_1.arg[i]} \leq \max_i (t_p^{T_1.arg[i]} + t_{cb}^{T_1.arg[i]}), \qquad \text{for } 1 \leq i \leq T_1.narg,$$

where $t_p^{T_1.arg[i]}$ is the processing time of the ith predecessor node of node T_1,

$t_{cb}^{T_1.arg[i]}$ is the communication time of the result of ith predecessor to node T_1,

and $T_1.narg$ is the number of predecessors of node T_1.

This condition amounts to examining the critical path of the subgraph assuming the predecessor nodes are activated at the same time, and comparing it with the sum of the processing times of all the nodes in the subgraph. This condition permits us to make a local decision on whether to execute the subgraph as it is or to lump it into one node and execute it sequentially.

This step is illustrated in Figure 2. Figure 2a is a subgraph with low-level parallelism and tasks with large overhead, while in Figure 2b the nodes T_1, T_2, T_3 and T_4 have been lumped together into a single node T_1'. In the subgraph of Figure 2a, node T_1 can execute only after the results from node T_2 and T_3 and T_4 have arrived. For the local optimization if we assume that nodes T_2, T_3 and T_4 are activated at the same time, then the result from nodes T_2 and T_3 will arrive after 5 cycles and the result from node T_4 will arrive after 8 cycles. Hence node T_1 is activated only after 8 cycles. In the sequential case (Figure 2b) the result from nodes T_2, T_3

and T_4 are available after 6 cycles, as we do not have to communicate between different processors. In this case the subgraph of Figure 2a can be reduced to Figure 2b.

The criterion for reduction is tested by an algorithm (*Upreduction* [19]) which traverses the graph starting at the exit node. It combines a node with its predecessors whenever the reduction criterion is met.

The reduction algorithm results in graph transformations of two kinds:

- Reduction of fine grain parallelism when local criterion suggests that it cannot be effectively utilized. These are transformation of the kind illustrated in Figure 2.

- Combination of sequential nodes. Sequential nodes which have single arguments and results are combined together into a single node, saving the communication time between them. This is illustrated in Figure 3.

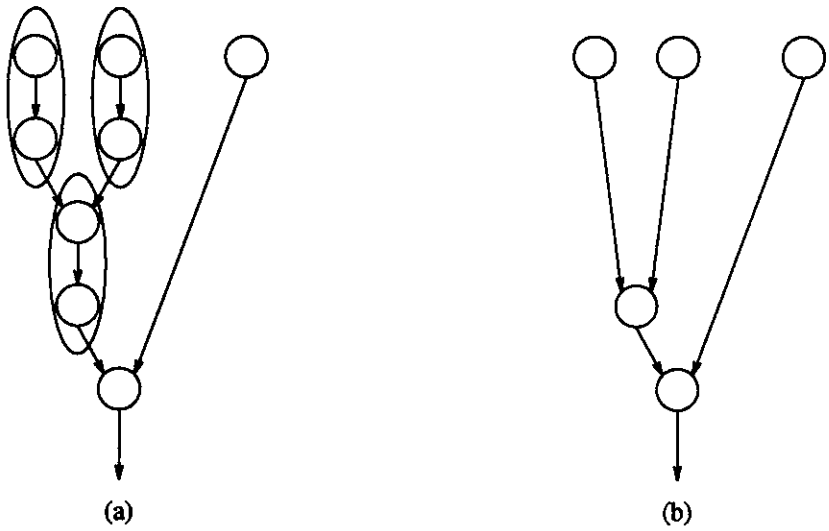

Figure 3: Reduction of Sequential Nodes

However, it must be emphasized that graph reduction is a heuristic, which is applied locally to a node assuming that all its predecessors can be activated at the same time. It can result in a longer critical path for the program when a node on the critical path is combined with predecessors not belonging to that path.

4. PERFORMANCE

To study the performance of the algorithm we chose 10 precedence graphs abstracted from real programs, which were allocated to varying number of processors, and statistics collected. Five of the precedence graphs chosen are irregular and asynchronous with arbitrary patterns of precedence. The other group of graphs is highly regular, symmetric and synchronous.

The five asynchronous graphs were obtained from Martin [18] with backward branches eliminated. The regular and synchronous graphs chosen are the 16-point Fast-Fourier Transform (FFT), Sort-Merge (SM), GRSEQ, Matrix Multiply (MM) and SYN5. The Sort-Merge graph is characterized by binary branching till a concurrency of 32 nodes and then a binary merge. GRSEQ is a group-sequential task graph with precedences from stage to stage. The 4X4 Matrix Multiply is obtained from a program written in Functional Programming Language (FPL). The final graph SYN5 is the same precedence graph as the last graph (ASYN5) of the previous group, but this time with constant and equal times instead of random times. This graph branches into a number of identical streams but the precedences within a stream are highly irregular. Table I summarizes the graphs used as benchmarks.

TABLE I

Name	Description	No. of Nodes
ASYN1	Weather Prediction (WWP 32)	32
ASYN2	Assignment & Sequencing (82V)	82
ASYN3	Complex Numerical Weather Problem (NWP 147)	146
ASYN4	Complex Assignment & Sequencing (L2)	193
ASYN5	Graph with identical streams (X-RAY)	223
FFT	16-point Fast-Fourier Transform	80
SM	Sort-Merge (Max. Parallelism =32)	94
GRSEQ	Task Graph with Stagewise Precedences	62
MM	4X4 Matrix Multiply	188
SYN5	ASYN5 with equal times	223

For the graphs in the synchronous group we assign equal times to each of the nodes and to each of the arcs. The processing and communication times for nodes of the asynchronous graph are chosen to be unequal

and vary over a range from the mean in order to investigate the performance for general program graphs. The execution time of each node (t_p), the local communication time (t_{cl}) and the bus communication time (t_{cb}) are chosen randomly from a uniform distribution across a specified interval. In our measurements we specified a range of 10% of the mean for the processing time and the bus communication time and kept the local communication time constant. The choice of a range of 10% along with the general asynchrony of the graph permits us to test our algorithm for graphs where nodes do not lie in distinct levels.

The principal performance characteristic is the *response or completion time* of the program graph, which we attempt to minimize. The response time is compared to the upper critical path length (t_{CP_H}) and the lower critical path length (t_{CP_L}), where

$$t_{CP_H} = \sum_{i \in \text{nodes on the critical path}} (t_p{}^i + \sum_{j \in \text{arcs from node } i} t_{cb}{}^{ij}) \text{ and}$$

$$t_{CP_L} = \sum_{i \in \text{nodes on the critical path}} (t_p{}^i + \sum_{j \in \text{arcs from node } i} t_{cl}{}^{ij}).$$

The other performance measures of interest are the *total bus communication time* and *average idle time* per processor. The total bus communication time is the sum of the times spent by each processor in communicating with other processors. The average idle time is calculated as the sum of the idle times of all processors divided by the number of processors (N). In the model of architecture we have considered the processor is busy during communication and hence communication times are not counted as idle times. Note that the idle time also includes the time from the completion of execution of tasks on a processor to the maximum completion time of all the processors.

The objectives of this performance evaluation are the following:

- To examine the behavior and characteristics of the allocation algorithm and its effectiveness in speeding up the computation in a multiprocessor system. In particular the response time is compared to the average idle time and the total bus time. We also study the effect of the variation of communication and

processing time on the performance of the allocation algorithm.

- Demonstrate the effect on response time of the two major features of our algorithm - precedence to critical tasks and minimization of communication time. In particular we are interested in comparing our algorithm to the critical path list schedules which have been reported [1, 15, 17] to be *near optimal*, where communication times have not been considered.

- Observe the improvement in response times when a fine grain graphs are reduced before allocation.

We performed more than 3000 allocations on the ten graphs, and present the general characteristics observed. We chose two representative graphs - Asynchronous X-RAY graph (ASYN5) and Matrix Multiply (MM) to illustrate our results.

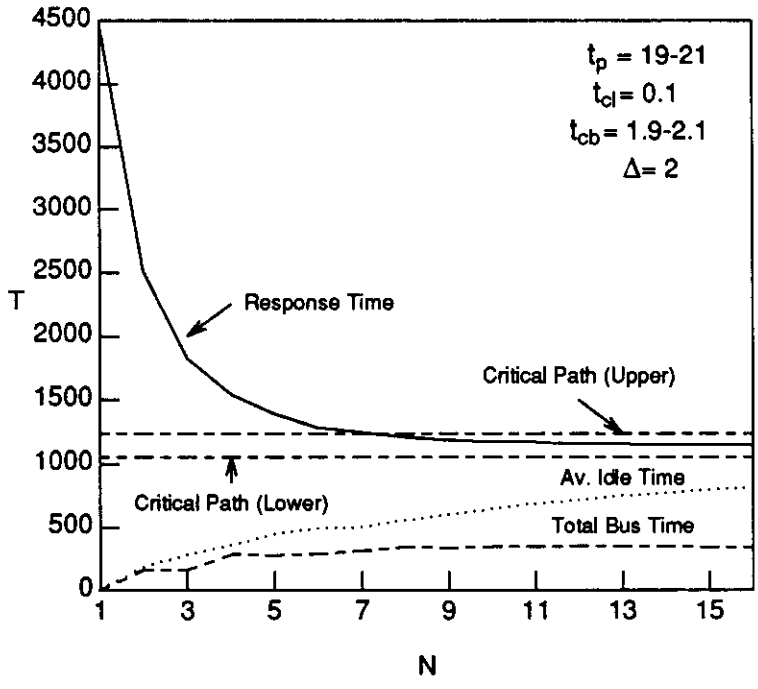

Figure 4 : Time Characteristics for Graph ASYN5

We first examine the reduction in response time of programs when allocated to multiprocessor systems. Figure 4 plots (for graph ASYN5) the variation of response time, average idle time per processor and the total time spent in bus communication with number of processors (N). Initially the speedup is almost linear, but as the number of processors is increased the response time saturates to a value between upper critical path length (t_{CP_H}) and the lower critical path length (t_{CP_L}). We observe that while the total bus communication time becomes constant, the average idle time increases as the number of processors is increased. Note that as the number of processors is increased, the average idle time per processor approaches the response time, implying very little processor activity in each processor.

From our observation, scheduling anomalies such as the increase in response time with additional processors are rare and cause negligible deviations. Their occurrence is most probable when the communication time to processing time ratio is very high, or when the response time curve has saturated and the number of processors is further increased. In synchronous graphs when the number of processors (N) is a factor of the total natural concurrency of the graph, then allocation to each processor is symmetrical and a significant improvement in response time is noted. However, when this happens in the saturated part of the curve and the number of processors is further increased then increases in the response time can be noted.

We next observe the effect of changing the ratio of bus communication time and processing time on the response time curves. Figure 5 compares the response time curves for different ratios of communication and processing time for the Matrix Multiply graph. The processing time is kept fixed while the bus communication time is varied. We keep the local communication time almost zero in order to isolate the change in bus communication time on the response time. Systems with extremely high communication time ratios (t_{cb}/t_p) are unable to utilize large number of processors and their minimum response times are several times higher than the lower critical path length (t_{CP_L}). But as the communication time becomes smaller than the processing time, the difference in response times for different ratios (for example t_{cb}/t_p = 1/5 & 1/10) becomes negligible. Thus, for

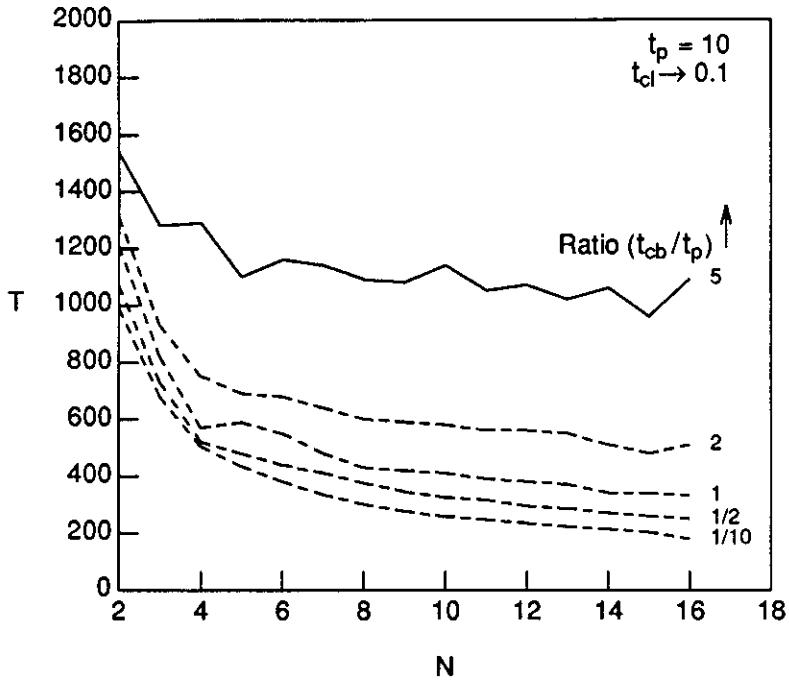

Figure 5 : Variation of Response Time for MM with Ratio of Communication and Processing Time

fine grain graphs the time taken by communication across processors limits the minimum response time achievable.

Our algorithm allocates a task to a processor based on two driving principles - precedence to critical tasks and the minimization of the communication time between this task and predecessor tasks which have already been allocated. We demonstrate next the importance of both principles and show how performance suffers when either one is ignored.

In Figure 6 we illustrate the performance improvement of our allocation algorithm over the case when only critical path scheduling is enforced and no attempt is made to have predecessor-successor tasks cohabit in the same processor. The examples are the Matrix Multiply and ASYN5 graph, where the ratio of communication time and processing time is 1/4. The curves in Figure 6 shows the percentage improvement in response time of

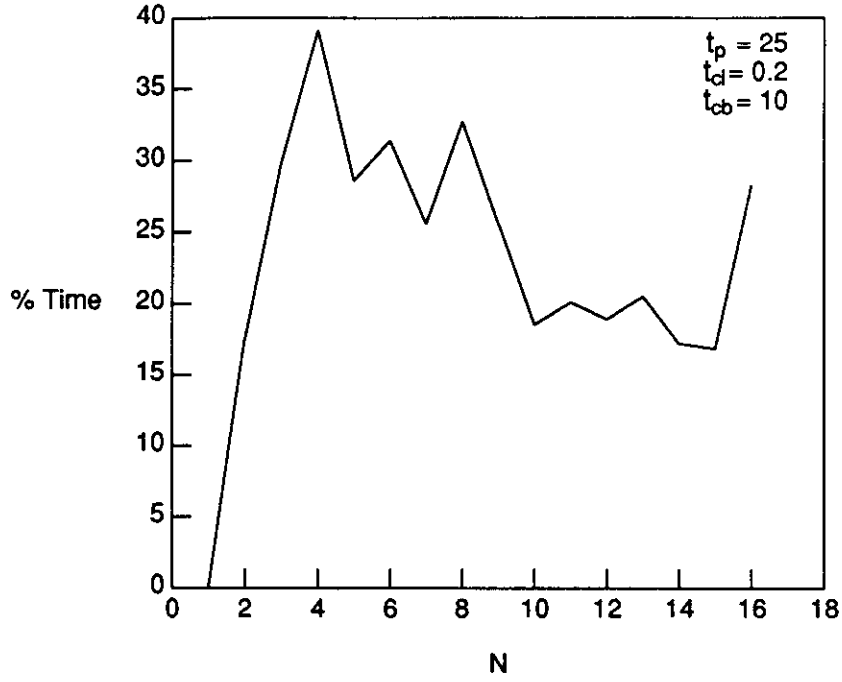

Figure 6a : Percentage degradation in Performance for Matrix Multiply by not enforcing minimization of communication criteria

our algorithm over the case when only critical path scheduling is enforced. We observe an average degradation of 24.7% for the Matrix Multiply graph and 6.9% for ASYN5, when no attempt is made to allocate a task and its successors to the same processor.

TABLE II

Graph	% Improvement in Response Time		
	t_{cb}/t_p = 1/10	t_{cb}/t_p = 1	t_{cb}/t_p = 2
ASYN1	1.9	9.9	13.8
ASYN2	0	3.1	1.2
ASYN3	0.9	9.6	10.6
ASYN4	0.2	2.4	3.3
ASYN5	1.2	12.6	18.9
FFT	1.6	15.8	27.1
SM	3.7	26.8	46.4
GRSEQ	1.2	16.5	38.4
MM	6.1	42.3	75.7
SYN5	2.0	21.4	27.1

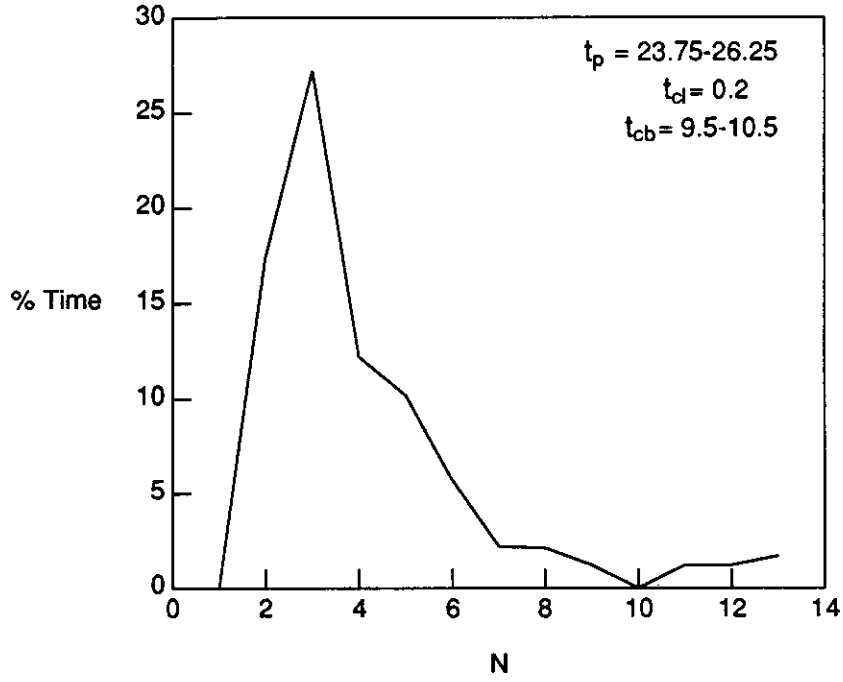

Figure 6b : Percentage degradation in Performance for ASYN5 by not enforcing minimization of communication criteria

Table II shows the percentage improvement in response time of our algorithm compared to the critical path allocation algorithm. For each of the five synchronous and asynchronous graphs we calculate the average improvement in response time for number of processors till saturation, for low and high ratios of communication and processing time.

As expected the improvement in response time is greater when the communication times are higher. This is because the net reduction in communication time achieved by reducing interprocessor predecessor-successor task communication is larger when the bus communication time is high. Also the communication minimization criterion has more effect for the synchronous graphs, because several tasks with the same critical path become enabled at the same time, and the choice of which task to allocate can be based on maximum saving in communication time, rather than in random. Graphs ASYN2 and ASYN4 have limited parallelism, and

have few candidates to choose from while allocating a task to a processor. Hence the additional communication reduction hueristic results only in a small gain over the performance achieved by the critical path criterion. We have also observed that the improvement is largest when the speedup is almost linear and reduces when the speedup curve saturates.

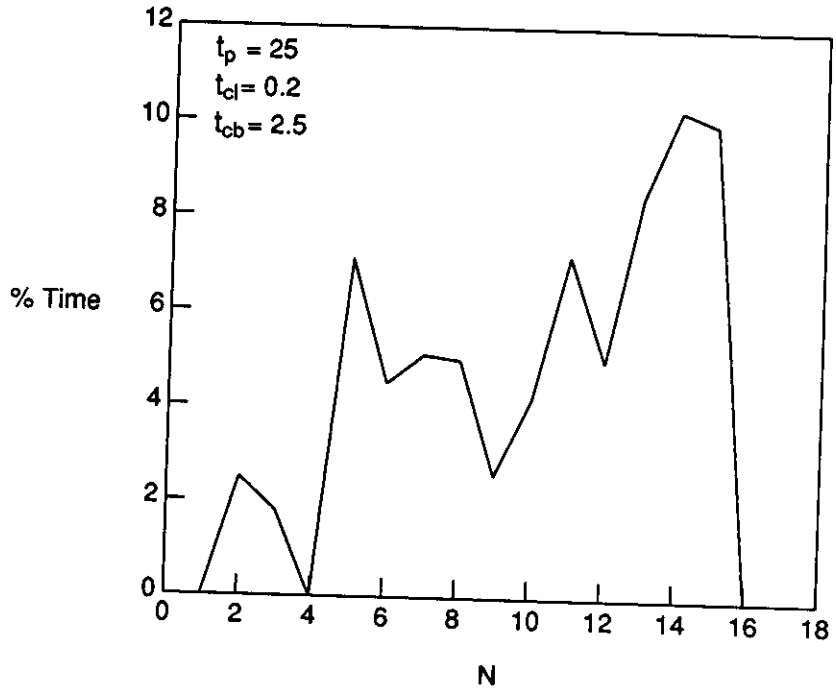

Figure 7a : Percentage degradation in Performance for Matrix Multiply by not enforcing critical path ordering

When the deviation (Δ) is made very large then the critical path list ordering is no longer operative. Tasks are allocated based only on the minimization of communication time. In Figure 7, we show the percentage improvement in performance of the algorithm compared to the case in which the only driving principle is the minimization of communication time. We show the improvement in response time when Δ=0 compared to when Δ=100 units for the Matrix Multiply, and Δ=10 compared to Δ=468 for ASYN3. The graph ASYN5 was not chosen to illustrate the effect of the critical path criterion, because lengths of parallel paths in ASYN5 do not differ sufficiently, and the average improvement is only 2.3 %. With the deviation (Δ) so high, at each stage of

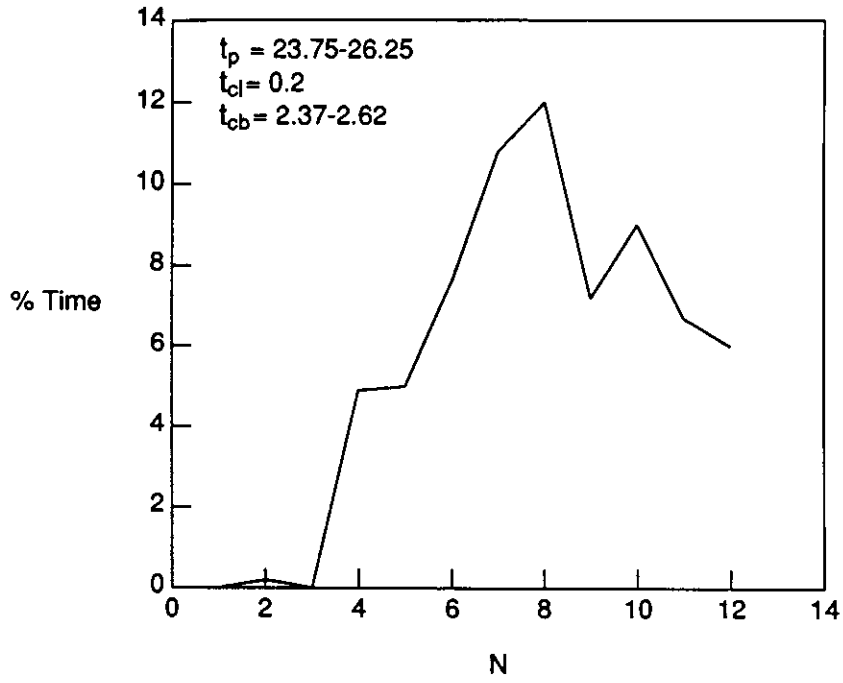

Figure 7b : Percentage degradation in Performance for ASYN3 by not enforcing critical path ordering

the algorithm the candidates for allocation to a processor are all the enabled tasks in the graph. For Matrix Multiply with Δ=0 the algorithm performs better than when Δ=100 units, except at certain points like N= 4 and 16 where the two do the same, because N is a factor of the natural concurrency of the graph, and in these cases the minimization of communication rule is sufficient to force symmetrical allocation. Also note that the critical path criterion has a more pronounced criterion when the number of processors increases.

Finally we examine the effect on performance of graph reduction prior to allocation. We observe that when bus communication times are high (t_{cb}/t_p > 2) then graph reduction which consists of lumping of sequential nodes and the combination of a node with its predecessors (reduction of parallelism), leads to significantly better performance. In Figure 8 we compare the response time curves for Matrix Multiply and ASYN5, with and without reduction for t_p = 10, $t_l \equiv 0$ and t_{cb} = 20 (in the average for the asynchronous graph), so that t_{cb}/t_p = 2. We observe an average improvement in the response time of about 16.1% for Matrix Multiply

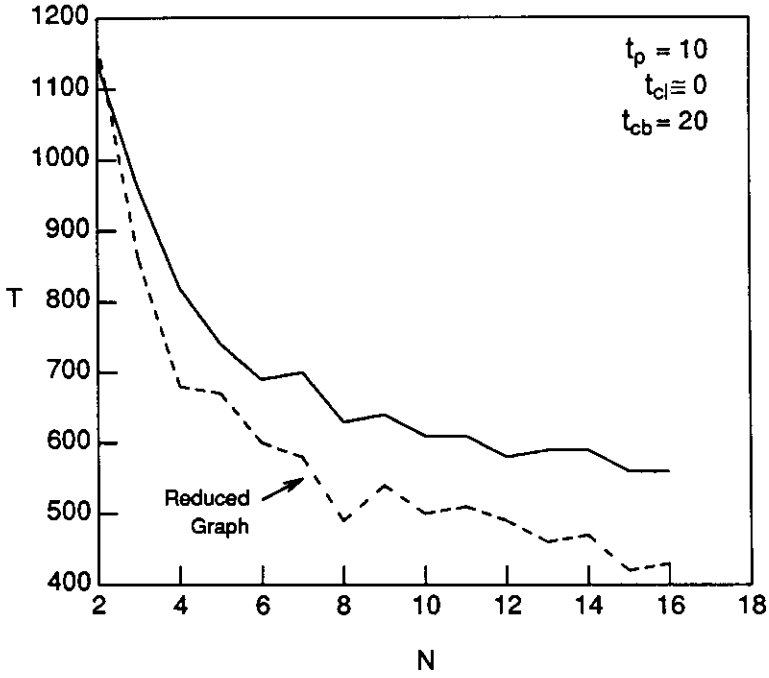

Figure 8a : Effect of Graph Reduction on Response Time for MM

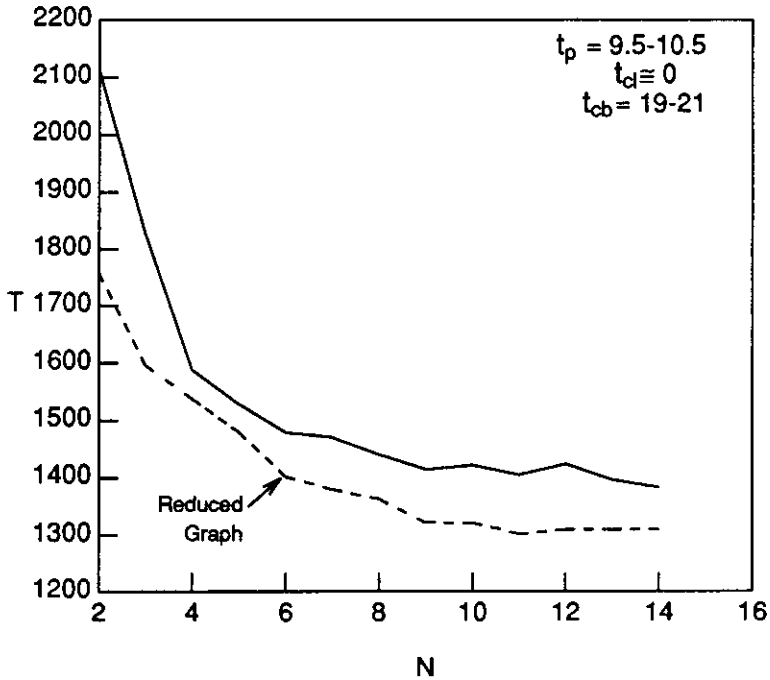

Figure 8b : Effect of Graph Reduction on Response Time for ASYN5

and 7.2% for ASYN5, when graph reduction is performed under these circumstances. Moreover the number of nodes to be allocated reduces from 188 to 106 for Matrix Multiply and from 223 to 134 for ASYN5, thus reducing the time for allocation. In Figure 9 we show the percentage improvement in response time of the graphs with just combination of sequential nodes and with both sequential reduction and reduction of parallelism.

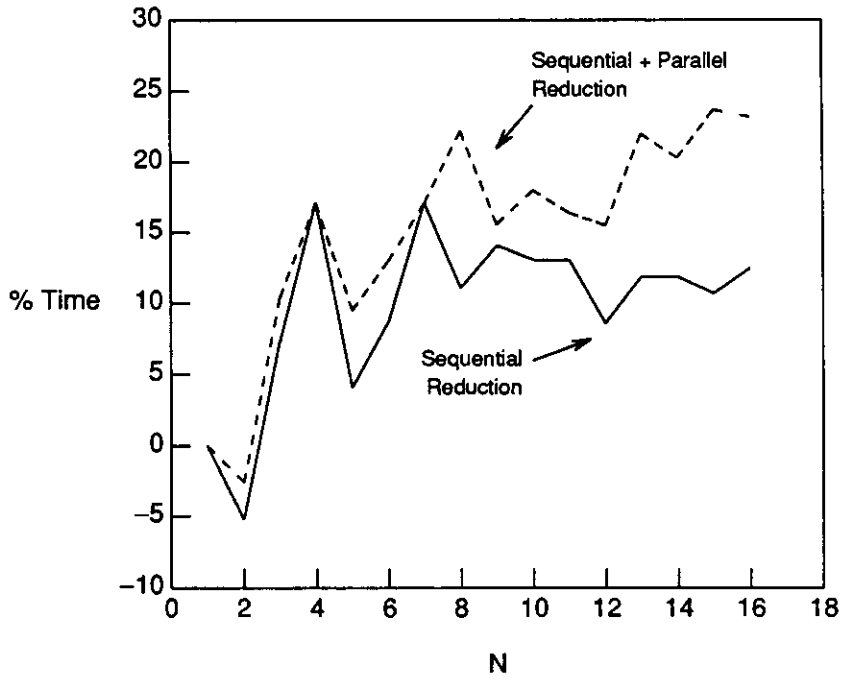

Figure 9a : Effect on Performance for MM due to Sequential Reduction & Both Sequential and Parallel Reduction

However when the processing times are much greater than bus times then reduction can cause a fall in performance. This can be attributed to the fact that we loose flexibility in allocating tasks when several tasks are lumped together. We studied the response time of the benchmark graphs with and without reduction, with t_p = 10, $t_{cl} \cong 0$ and $t_{cb} = 1$ (in the average for the asynchronous graphs). For such a low ratio of communication and processing times, only sequential reduction took place, and hence two of the ten graphs remained unchanged on reduction. The maximum degradation in response time observed in our experiment was 6.7 %. In only 55 % of the cases was there any change in the response time when reduction took place, and the average net increase in

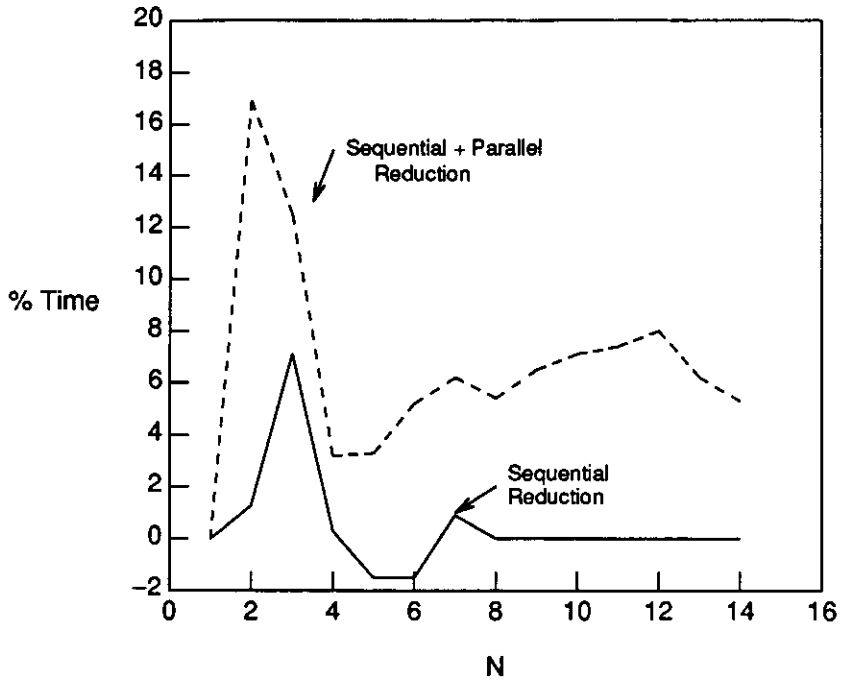

Figure 9b : Effect on Performance for ASYN5 due to Sequential Reduction & Both Sequential and Parallel Reduction

response time when a change was observed was only 0.4 %. We thus note that even when the tasks have low control and communication overhead, the penalty of preprocessing the graph is minimal.

5. CONCLUSIONS

We have developed an algorithm for allocation of a acyclic graph data flow graph to a general multiprocessor system. It is a list scheduling algorithm which incorporates variable communication times, depending on whether the successor task is allocated to the same or different processors. The algorithm is based on two principles:

- Precedence to critical tasks
- Minimization of communication time

Performance studies indicate that our algorithm is well behaved, relatively anomaly free, and compares favorably with critical path algorithms which did not take any special measures to handle communication time.

When the program graph is fine grain, i. e. the control and communication overhead is comparable to or exceeds the processing time of a task, then we preprocess the graph with a graph reduction heuristic. The graph reduction algorithm forces the tasks of the graph to be of appropriate size before allocation. From experiments we observe that graph reduction leads to significant improvement in performance if the graph is fine grain, i.e. when the ratio of communication time and processing time is large. However with large grain task graphs, graph reduction is not effective.

ACKNOWLEDGEMENTS

This work was supported in part by the Contract No. 25-3074 "Multiprocessor System Evaluation and Programming Environment" from the Sandia National Laboratories. We are grateful to Mr. George Davidson of Sandia National Laboratories for his comments and cooperation in this work.

REFERENCES

1. Adams, T.L., Chandy, K. M., and Dickson, J. R. A comparison of list schedules for parallel processing systems. *Comm. ACM* 17, 12 (Dec. 1974), 685-690.

2. Arvind and Culler, D. E. Tagged Token Dataflow Architecture. Tech. Rep. 229, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, Mass., July 1983.

3. Babb II, Robert G. Parallel processing with large grain data flow techniques. *IEEE Comput.* (July 1984), 55-61.

4. Borgman, C. R. and Pierce, P. E. A hardware/software system for advanced development guidance and control experiments. *AIAA Computers in Aerospace Conference*, AIAA-83-2416, Oct. 1983, Hartford, CT, 377-384.

5. Davidson, G. Personal communication, Oct. 1985.

6. El-Dessouki, O., Huen, W. and Evens, M. Towards a partitioning compiler for a distributed computing system *Journal of Digital Systems* V, 1/2 (1981), 157-179.

7. Ercegovac, M. D., Chan, P. K. and Ravi, T. M. A dataflow multiprocessor architecture for high speed simulation of continuous systems. *Proc. International Workshop on High-Level Architecture*, 1984.

8. Ercegovac, M. D. et al. Task partitioning, allocation and simulation for a dataflow multiprocessor system. *Proc. Summer Computer Simulation Conference*, 1984.

9. Gajski, D. D. et al. Cedar. *Proc. Compcon*, Spring 1984, 306-309.

10. Gajski D. D. and Pier, Jih-Kwon. Essential Issues in Multiprocessor Systems. *IEEE Comput.* (June 1985), 9-27.

11. Gaudiot, J. L. On program decomposition and partitioning in data-flow systems. UCLA Computer Science Department Report No. CSD-821212, Dec. 1982.

12. Gaudiot, J. L. and Ercegovac, M. D. Performance evaluation of a simulated data-flow computer with low resolution actors. *Journal of Parallel and Distributed Computing*, **2**, 321-351 (1985).

13. Harris, D. L. Inter-processor communication. Sandia Aerospace Computer Development (SANDAC), Sandia National Laboratories, Albuquerque, New Mexico, May 1984.

14. Hwang, K. and Su, S. P. Priority scheduling in event-driven dataflow computers. TR-EE 83-36, School of Elec. Eng., Purdue Univ., Dec. 1983.

15. Kohler, Walter H. A Preliminary evaluation of critical path method for scheduling tasks on multiprocessor systems. *IEEE Trans. Comp.* (Dec. 1975), 1235-1238.

16. Lenstra, J. K. and Rinnooy Kan, A. H. G. Complexity of scheduling under precedence constraints. *Operations Research* **26**, 1 (Jan.-Feb. 1978), 22-35.

17. Lord, R. E. Scheduling recurrence equations for solution on MIMD type computers. Ph.D. Dissertation, Washington State University, Pullman, WA, 1976.

18. Martin, D. and Estrin, G. Experiments on models of computations and systems. *IEEE Trans. on Electronic Comp.* (Feb. 1967), 59-69.

19. Ravi, T. M. and Ercegovac, M. D. Allocation for the Sandac multiprocessor system. UCLA Computer Science Department Report No. CSD-860059, Feb. 1986.

20. Ravi, T. M. Partitioning and Allocation of Functional Programs for Data Flow Processors. M.S. Thesis, UCLA Computer Science Department Report No. CSD-860063, Apr. 1986.