PAPER

Scheduling DAGs on Message Passing m-Processor Systems

Sanjeev BASKIYAR†, *Nonmember*

SUMMARY Scheduling directed a-cyclic task graphs (DAGs) onto multiprocessors is known to be an intractable problem. Although there have been several heuristic algorithms for scheduling DAGs onto multiprocessors, few address the mapping onto a given number of completely connected processors with an objective of minimizing the finish time. We present an efficient algorithm called *ClusterMerge* to statically schedule directed a-cyclic task graphs onto a homogeneous completely connected MIMD system with a given number of processors. The algorithm clusters tasks in a DAG using a longest path heuristic and then iteratively merges these clusters to give a number of clusters identical to the number of available processors. Each of these clusters is then scheduled on a separate processor. Using simulations, we demonstrate that *ClusterMerge* schedules task graphs yielding the same or lower execution times than those of other researchers, but using fewer processors. We also discuss pitfalls in the various approaches to defining the longest path in a directed a-cyclic task graph.

key words: *clustering, DAG, longest path, multiprocessors, non-preemptive scheduling*

1. Introduction

Scheduling tasks* onto a multiprocessor system is paramount to the efficient execution of programs and utilization of the system. One may schedule the tasks onto processors at compile time, called static scheduling or at run-time when it is called dynamic scheduling. Dynamic scheduling can adapt to the changing resource requirements at run-time but has substantial overheads primarily due to task relocation and running the scheduling algorithm itself at run-time. On the other hand, most of the overhead in static scheduling occurs at compile time. Often a dynamic scheduling algorithm is used to fine tune the schedule after a static schedule has been constructed. In this paper, we restrict our attention to static scheduling.

We define a DAG as an a-cyclic graph with nodes representing tasks and edges execution precedence between tasks. To each node and edge of the graph is associated a node weight and an edge weight. The node weight represents the serial execution time of the task and the edge weight represents the data communication time between the connecting tasks, if the tasks are executed on adjacent processors of a multiprocessor system. If the tasks are executed on the same processor the communication time is zero.

We address the problem of scheduling weighted directed a-cyclic task graphs onto completely connected homogeneous distributed memory multiprocessor systems with a given number of processors. Many researchers address the problem of mapping programs on an unbounded number of processors, but the availability of a large number of processors is a luxury not available to most users. Although circuit fan-in/fan-out requirements prohibit the construction of completely connected systems with a large number of processors, it may be possible to construct completely connected systems with a relatively small number of processors. Also, the proliferation of multiprocessor workstations is expected to continue into the next century thereby extending the accessibility of such systems. Our schedule is also useful in cases when a job uses only a clique of a massively parallel processor system, where the clique consists of a small number of completely connected processors. Furthermore, it may be possible to use our algorithms (when followed by a cluster allocation mechanism) on systems that are not completely connected, with good execution times.

The problem of scheduling directed a-cyclic task graphs on a fixed number of processors to minimize the finish time in the absence of inter-processor communications is NP-complete [11]. Also, the problem of scheduling directed a-cyclic task graphs to minimize the finish time on an unbounded number of processors [9] in presence of inter-processor communications is NP-complete. Although several heuristic scheduling algorithms have been proposed to schedule task graphs onto parallel machines, few address the issue of minimizing the *maksespan* or the finish time and even fewer account for the overlap of computation and communication times on machines with a given number of processors. In this paper we have developed our algorithm keeping the above issues in mind. We manage the mapping problem by dividing it into steps of clustering tasks, merging the clusters, allocating the clusters onto processors and scheduling the execution of tasks within the merged clusters. One may observe that the intractability of the problem is reflected in the interdependence of these steps. Interestingly, a benefit of

Manuscript received March 3, 1999.
Manuscript revised November 22, 1999.

†The author is with the Department of Computer Science and Engineering, Auburn University, Auburn, AL 36849.

*A task is a unit of code whose execution, once initiated, runs to completion without wait upon any data external to the task.

the target system being completely connected is that it makes the step of allocation of clusters trivial thereby eliminating an interdependence step which somewhat simplifies the problem.

The organization of the remainder of this paper is as follows. In the next section we review previous research in this area, in Sect. 3 we describe the target architecture for scheduling DAGs and present a non-preemptive scheduling algorithm to minimize the finish times of directed a-cyclic task graphs. In Sect. 4 we show simulation results and compare the performance against those in [6],[12] and in Sect. 5 we make concluding remarks.

2. Background

Extensive work has been done on scheduling tasks on parallel systems: almost exhaustive surveys appear in [2] and [7]. Clustering techniques [3] that group tasks into clusters based upon certain criteria and assign each cluster to a different processor have been shown to give good schedules. Sarkar's clustering algorithm [10] proceeds by first assuming that all tasks in a task graph are executed on different processors. It sorts the edges of the task graph in descending order of their communication times. Next, it merges the tasks connected by the edge with the highest communication time if doing so does not increase the finish time. The algorithm completes upon scanning all edges. Sarkar assumes the availability of an unbounded number of processors. Kasahara and Narita [5] assign the highest priorities to tasks in the critical path; tasks not in the critical path are assigned a priority equal to the number of successors. Once the priority list is constructed scheduling is performed using a list-scheduling algorithm: tasks are dispatched, based upon their priorities to the first available processor. Lo [8] uses the network flow model: she does not consider the overlap of communication with computation. Yang and Gerasoulis' Dominant Sequence Clustering (*DSC*) algorithm [12] provides a schedule within a factor of two of optimal for coarse grain directed a-cyclic task graphs on an unbounded number of processors. However, our scheduling algorithm generates a schedule based upon the number of available processors specified by the user. The *DSC* algorithm defines the *tlevel* of any task i, as the length (excluding the execution time of i) of the longest path from the entry task to task i, and its *blevel* to be the length of the longest path from i to the exit task. The priority of a task is defined to be the sum of its *blevel* and *tlevel*. Every task in the beginning is assumed to constitute its own cluster. The algorithm successively finds tasks i with the highest priority and merges it with the cluster of its predecessor that decreases $tlevel(i)$ maximally; if none of the possible mergers decrease $tlevel(i)$, a merging is not performed. After each merging, the priorities of the successors of the merged tasks are updated. Yu's [13] algorithm assigns nodes in a critical path to a unique cluster, deletes the critical path and repeats the above process on the remaining graph until it becomes empty. Yu imposes an upper bound k on the number of nodes of a critical path that can be assigned to a cluster, apparently to balance the load on the processors. However, this is not helpful—because of the precedence constraints, the nodes that could not be included in the selected k nodes of the critical path cannot be executed until the selected k nodes have been executed. Papadimitriou and Yannakakis [9] establish a bound for their scheduling heuristic, based on the maximum communication delay between any two tasks. They allow task duplication, whereas we do not allow task duplication in our algorithm. Also, Kruatrachue and Lewis' algorithm [2] uses task duplication. Task duplication requires more memory—a luxury not affordable in many real-time systems. Kim and Browne [6] transform a DAG into a Virtual Architecture Graph (VAG) that consists of clusters of tasks. This involves successively finding task clusters consisting of linear paths in a DAG with maximal costs. The cost of a path is a defined function of task and edge weights of the path and those of edges adjacent to the path. These clusters are then merged based upon the level number of their tasks (the level number of a task is defined to be one more than the maximum level number of its ancestors with the root having the level number of unity). Two clusters are merged if they do not have tasks with identical level numbers and have strong sequential dependence. Also, if the path of the maximum schedule length spans more than one cluster, clusters are iteratively split and merged. For the VAG, the initial target architecture is a completely connected multiprocessor system although mapping to generalized architectures is also addressed.

3. Scheduling Algorithm

In this section, we present a heuristic algorithm called *ClusterMerge* for non-preemptively scheduling a directed a-cyclic task graph on a completely connected distributed memory multiprocessor system composed of m identical processors. In the target system the processors communicate with a non-blocking send protocol. To receive data a processor has to poll its input buffer. A processor can not interrupt another processor for data. We assume that the input and output buffers of processors are large enough to buffer all messages.

The following are the inputs to the algorithm *ClusterMerge*: a directed a-cyclic task graph (DAG), G with a single root node, the number, m, of completely connected processors available for executing G. Also available are $w(i)$ and $c(i,j)$ that represent the weights of any node i and edge (i,j) in the DAG. The scheduling algorithm *ClusterMerge* consists of two parts: *ClusterNodes* and *MergeClusters*. The algorithm *Cluster*

Nodes groups nodes of G into clusters. It finds the longest linear path, L, in G and adds it to a set of clusters, C. Next the longest path is deleted from G. Any disjoint linear paths in G are also deleted and added to C. The above operations are repeated until G becomes empty. If the number of clusters obtained by *Cluster Nodes* exceeds the number of processors m, *MergeClusters* merges clusters in C to yield a number of clusters identical to the number of processors, i.e., $|C| = m$. The clusters to be merged are heuristically chosen to minimize the execution time of G.

3.1 Algorithm *ClusterNodes*

The algorithm *ClusterNodes* needs to identify the longest execution path in a DAG. Therefore the method to find the longest execution path is important lest we may attempt to minimize the execution time of a non-critical path. Clearly, because of precedence constraints, nodes in a path in a DAG can not be executed until all its predecessors have executed. Therefore it makes sense to assign all nodes in a path to a single processor; doing so also eliminates the communication time among the nodes in the path. We therefore define the longest path in a DAG as the path from a leaf to the root node that takes the longest execution time among all other such paths if all its nodes are assigned to execute on a single processor†. However, finding such a path is hard, as we see in the following attempts at establishing a method to find such a path.

(i) Find the path from a leaf node to the root node for which the sum of the node weights is maximal of all other such paths in the DAG. In Fig. 1, let $L = \{5, 4, 3, 2, 1\}$ be such a path. Consider another path $P = \{7, 6, 3, 2, 1\}$. Let the corresponding clusters $C_1 = \{5, 4, 3, 2, 1\}$ and $C_2 = \{7, 6\}$ be assigned to execute on processors M_1 and M_2. We observe that L may not take the longest time to execute. The inter-processor communication between P_1 and P_2 due to data communication over edge $(6, 3)$ may make the finish time for the path P being more than for path L.

(ii) Find the path from a leaf node to the root node for which the sum of the node and edge weights along the path is maximal over all other such paths. Let in Fig. 1, $L = \{5, 4, 3, 2, 1\}$ be such a path. Consider another path $P = \{7, 6, 3, 2, 1\}$. Although $\sum_P (w(i) + c(i, j)) \leq \sum_L (w(i) + c(i, j))$ it is possible that $\sum_P w(i) + c(6, 3) > \sum_L w(i)$ making the processing time for path P greater than for path L.

(iii) Find the path from a leaf node to the root node for which the sum of the node weights is maximal and there is no edge e incident on the path such that another path through e is longer. In Fig. 1, if $L = \{5, 4, 3, 2, 1\}$ be such a path, the following conditions must hold: $\sum_L w(i)$ is maximal of all such paths and $w(4)+w(5) \geq w(6)+w(7)+c(6, 3)$. Such a path may not exist in a DAG and therefore in an optimal assignment the path that finishes last may need to be processed in parts by more than one processor.

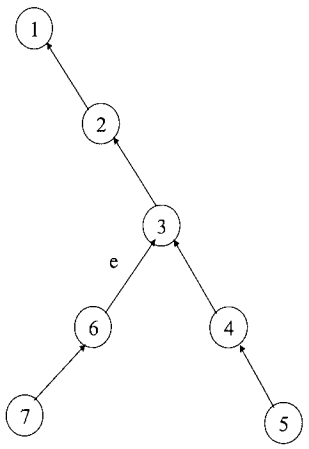


Fig. 1 The longest path.

Although the method in case (iii) above finds the longest path according to our definition, as pointed earlier, such a path may not exist in the DAG (particularly since *ClusterNodes* needs to find the longest path in the DAG several times). If we choose the method in case (i), we risk having high communication times. Therefore, we choose the method in case (ii) that selects the path having the maximal sum of node and edge weights††.

Next, we present the algorithm *ClusterNodes*. The following are the inputs, outputs and definitions in this context.

Inputs

N, the set of natural numbers.

$M = \{1, \ldots, m\}$, the set of completely connected identical processors.

$G \equiv (V, E, \propto)$ where

$V = \{1, \ldots, n\}$ is the set of vertices of G.

$\propto$ = a partial order on the elements of V.

$E = \{(i, j) :$ there is an edge from $i \in V$ to $j \in V\}$

†The critical path, in absence of inter-task communications, is defined in [4] as the path that finishes last; in this context the definition is independent of the assignment and the target topology. When inter-task communications come into play, one may define a critical path as one which finishes last but only for a *particular assignment* of a task graph and on a particular topology. This is so because inter-task communications have the peculiarity of vanishing when the tasks are assigned onto the same processor.

††Although a few other researchers use the same criterion for finding the longest path, the subtle rationale has not been pointed before. This rationale may have wide ramifications.

if $r \in V$ is the root node $Out(r) = \Phi$. //The function Out has been defined below
$w : V \rightarrow N$ and $c : E \rightarrow N$.
/* For any $i, w(i)$ is independent of the processor on which i is executed since all processors are identical */

Output

C, a set of clusters

Definitions

$In(i) = \{j \in V : (j, i) \in E\}$
$Out(i) = \{j \in V : (i, j) \in E\}$
$Leaf(i)$ iff $In(i) = \Phi$.
Path $P = \{p_1, p_2, \ldots, p_k\}$ where $\forall_{i=1}^{k} p_i \equiv V$ and $\forall_{i=1}^{k-1}(p_i, p_{i+1}) \in E$.
$Cost(P) = \sum_{i=1}^{k} w(p_i) + \sum_{i=1}^{k-1} c(p_i, p_{i+1})$
$C_k = \{i : i \in V\}, C_k \subseteq V$ is a cluster.
$C = \{C_k : C_k$ is a cluster$\}$ //Set of clusters
$G = \Phi$ if $V = \Phi$.

Algorithm *ClusterNodes*

int y /* Path suffix */

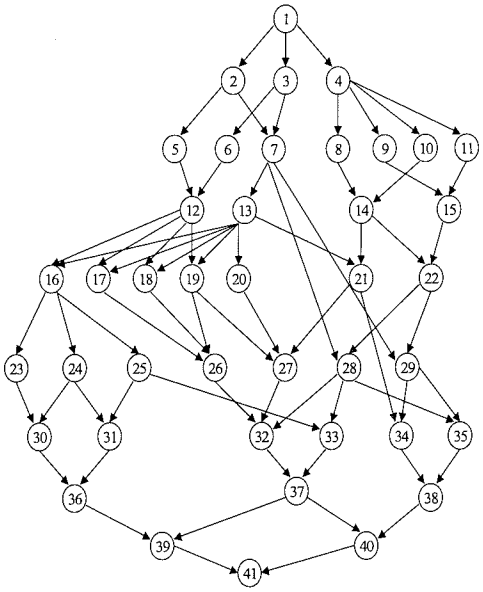


Fig. 2 Task graph T_1.

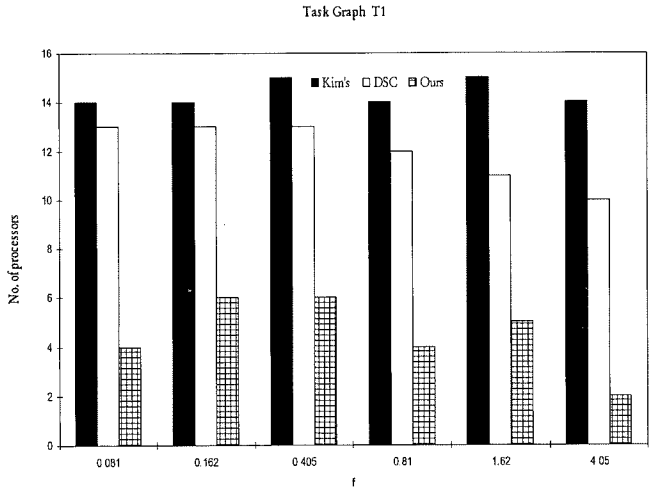


Fig. 4 Comparison of the number of processors needed to obtain the minimum execution times.

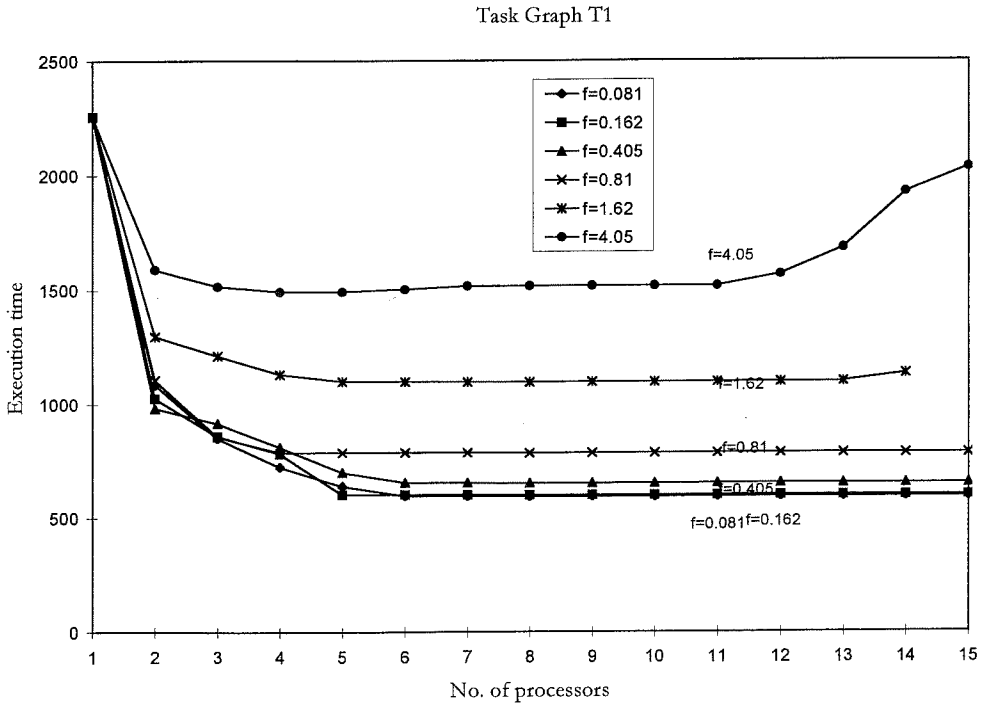


Fig. 3 Execution times using different number of processors.

$C \leftarrow \emptyset$ /* Initialize */
while $G \neq \emptyset$ **do**
/* Remove the longest path from G and assign it to a cluster */
Select path $P = \{p_1, p_2, \ldots, p_k\} \in G$ such that
(a) $Out(p_k) = 0$
(b) $In(p_1) = 0$
(c) $(p_i, p_{i+1}) \in E$, for $1 \leq i < k$, $p_i \in V$.
(d) $Cost(P)$ is maximal over V.
$V \leftarrow V - P$ /* Delete nodes of P from G */
$E \leftarrow E - \{(i,j),(j,i) : i \in V, j \in P\}$
/* Delete edges of G connected to P */
$C \leftarrow C \cup P$ /* Add path to C */
/* Remove all disjoint linear paths from G and add them to C */
Find all paths $P_y = \{p_1, p_2, \ldots, p_k\} \in G$ such that
(a) $|In(p_i)| = 0$, if $i = 1$
$= 1$, if $i \neq 1$
(b) $|Out(p_i)| = 0$, if $i = k$
$= 1$, if $i \neq k$
(c) $(p_i, p_{i+l}) \in E$, for $1 \leq i < k$.
Let the number of disjoint linear paths found above be Y.
for $y \leftarrow 1$ **to** Y **do**

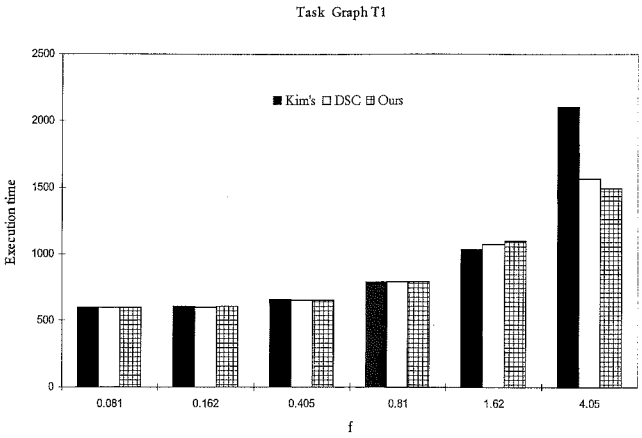


Fig. 5 Comparison of the minimum execution times.

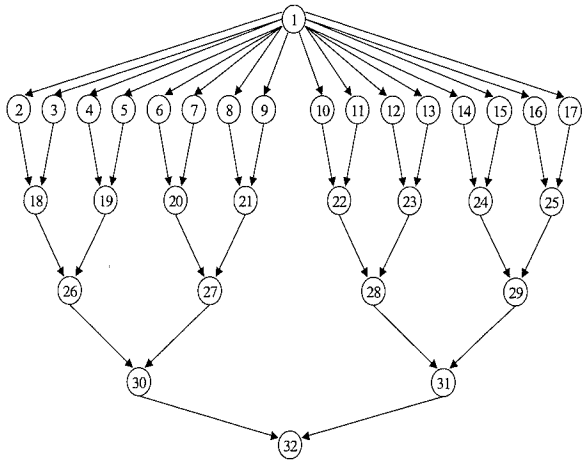


Fig. 6 Task graph T_2.

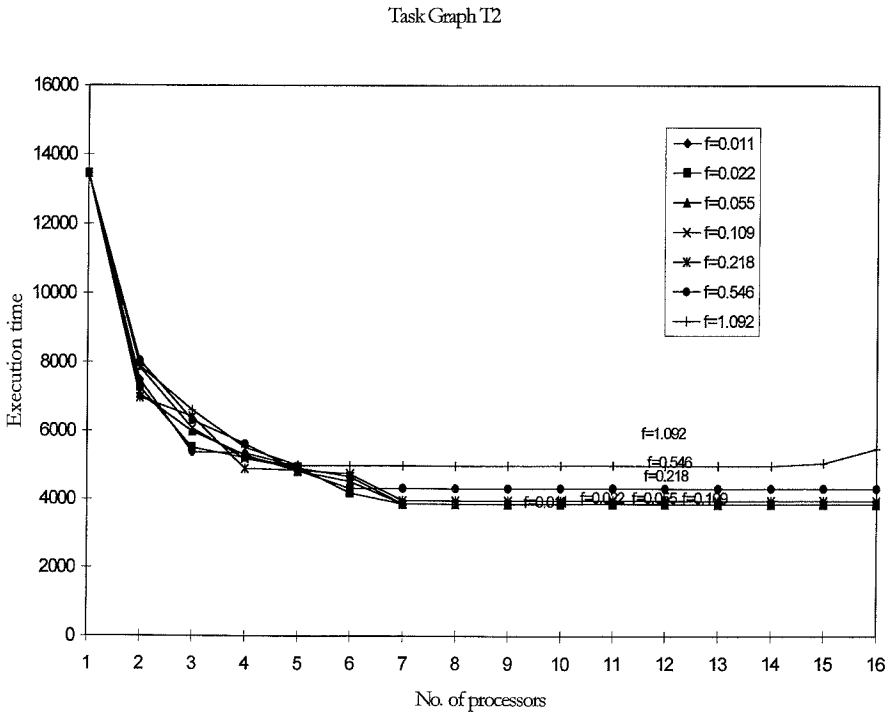


Fig. 7 Execution times using different number of processors.

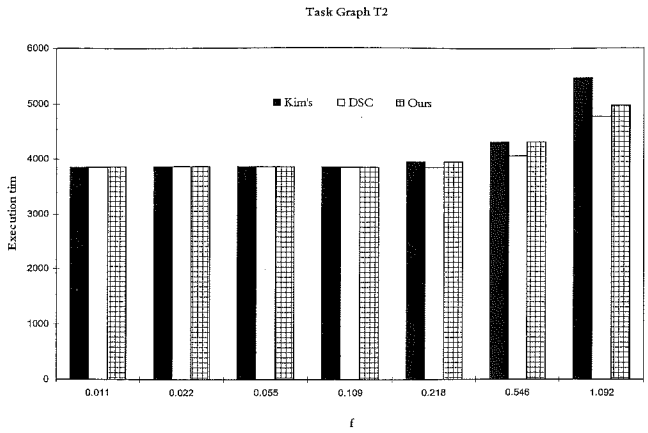


Fig. 8 Comparison of the number of processors needed to obtain the minimum execution times.

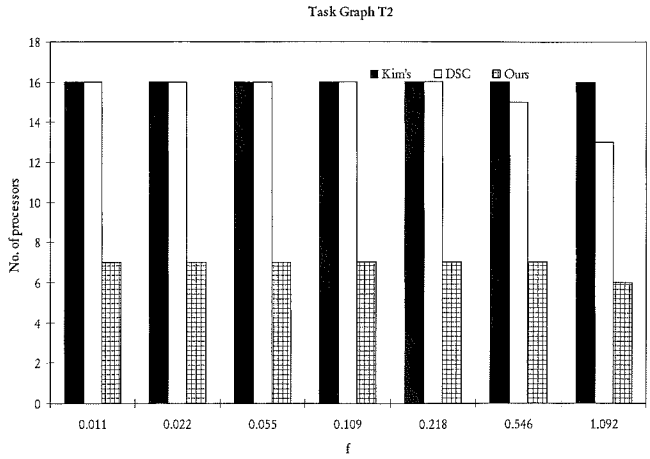


Fig. 9 Comparison of the minimum execution times.

$C \leftarrow C \cup P_y$ /* Add disjoint path */
$V \leftarrow V - P_y$ /* Delete nodes in P_y */
$E \leftarrow E - ((p_i, p_{i+1}) \in E, \forall_{i=1}^{k-1} p_i \in P_y)$
/* Delete edges in P_y */
endfor
endwhile
end *ClusterNodes*

The time complexity of *ClusterNodes* is $O(n + |E|)$.

3.2 Algorithm *MergeClusters*

Let u be the cardinality of the set of clusters C obtained from the algorithm *ClusterNodes*. If $u > m$, we must merge clusters in C to get m clusters so that each cluster can be assigned to a separate processor. Consider the time complexity of an optimal algorithm that uses an exhaustive search to determine the best clusters to be merged. Any cluster can be assigned to any of the m processors. Since all processors are identical, the number of possible assignments when $u > m$ is given by $S(u, m)$ the Sterling number [14] of the second kind. The assignment that gives the best completion time must be chosen. Clearly, the time complexity is very high. Therefore, in the algorithm *MergeClusters* we identify the clusters to be merged heuristically with an objective of minimizing the finish time of execution.

Before we present the algorithm *MergeClusters* we define two supplementary functions. The earliest event time [4] of a node $q \in V, ee(q)$, is the length of the longest of the paths to q from any of the leaf nodes. Let us define functions EE and F on a cluster and a set of clusters respectively. The function EE operates on any cluster $C_i \in C$ and orders the nodes in C_i in non-decreasing earliest event times. The function F operates on a set of clusters C of G and gives the completion time of G when each cluster in C is executed on a separate processor. A pair of clusters to be merged is chosen if merging them yields a lower finish time than merging any other pair of clusters.

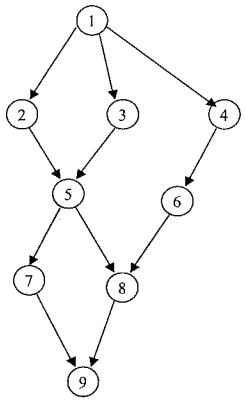


Fig. 10 Task graph T_3.

Algorithm *MergeClusters*
//Input : C, the set of clusters from *ClusterNodes*
int x, y, p, q // Cluster indices and loop counters
float T, t // Completion times of the DAG
Cluster $Temp, C_i$ // Set of nodes
ClusterSet $C = \{C_i\}, K$ // Set of clusters

If $m = 1$ **then**
/* Create a list of all tasks sorted
in non-decreasing order
of earliest event times */
$Temp \leftarrow \cup_{i=1}^{j} C_i$
$Temp \leftarrow EE(Temp)$
$C \leftarrow \{Temp\}$
$T \leftarrow F(C)$ // Compute the finish time
return
endif
repeat
$T \leftarrow \infty$
for $p \leftarrow 1$ **to** u
for $q \leftarrow p + 1$ **to** u
/* Temporarily merge C_p and C_q

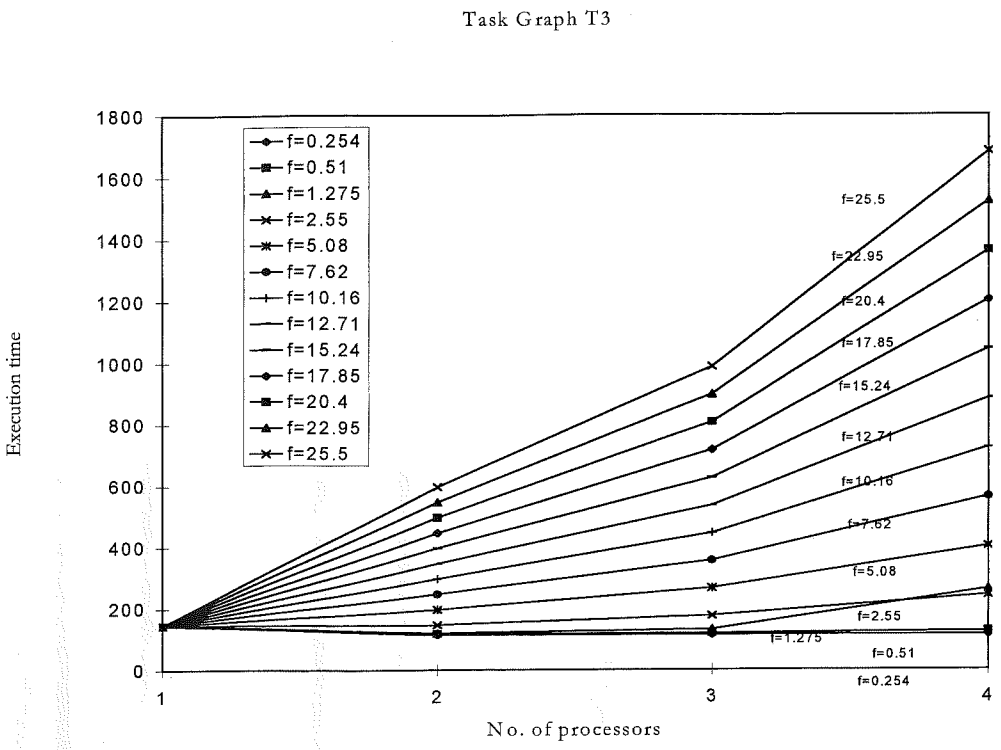


Fig. 11 Execution times using different number of processors.

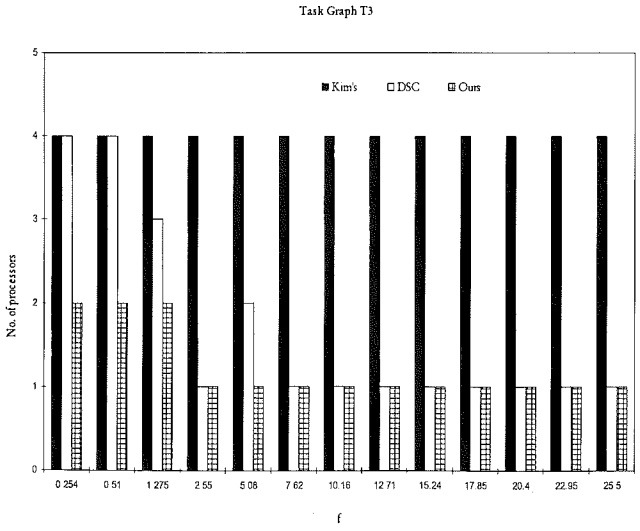


Fig. 12 Comparison of the number of processors needed to obtain the minimum execution times.

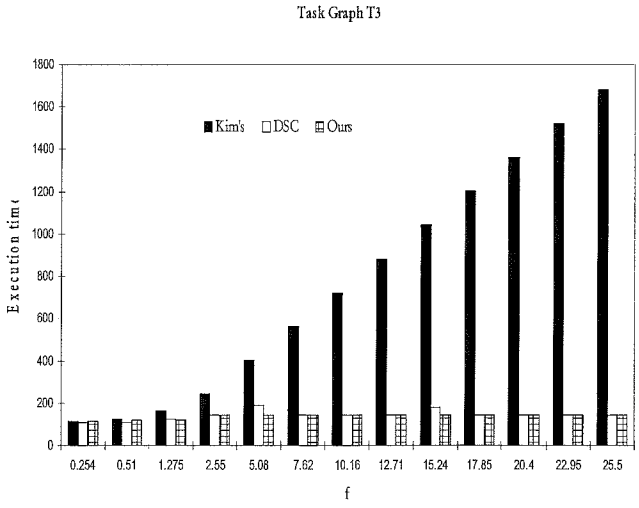


Fig. 13 Comparison of the minimum execution times.

and sort $Temp$ in non-decreasing order of earliest event times */
$Temp \leftarrow C_p \cup C_q$
$Temp \leftarrow EE(Temp)$
$K \leftarrow C \cup \{Temp\} - \{C_p\} - \{C_q\}$
//K has C_p and C_q merged
$t \leftarrow F(K)$
if $t < T$ **then**
$T \leftarrow t, x \leftarrow p, y \leftarrow q$ //Save cluster indices for future reference
endif
endfor
endfor
/* Merge clusters which gave the minimum execution time of all other pairs on merging */
$Temp \leftarrow C_x \cup C_y$
$C \leftarrow C \cup \{Temp\} - \{C_x\} - \{C_y\}$
$Temp \leftarrow EE(Temp)$
$u \leftarrow u - 1$ // Decrement the number of clus-

ters

until $u = m$ // No. of clusters equal no. of processors

end *MergeClusters*

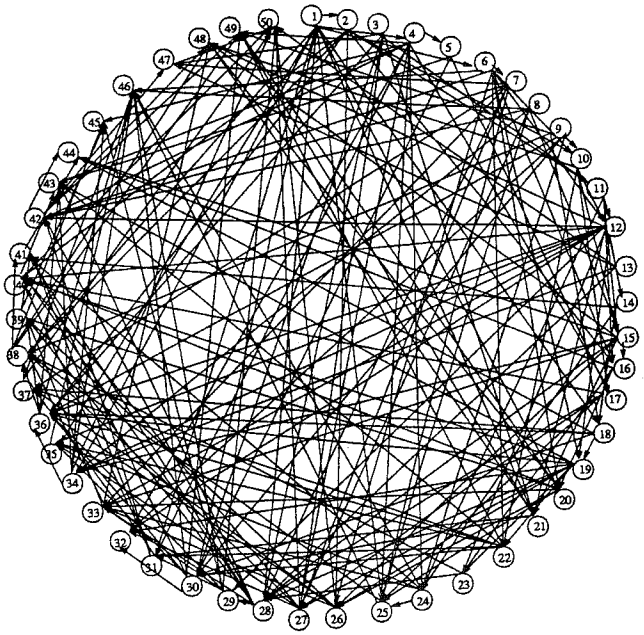


Fig. 14 Task graph T_4.

Output: Each cluster $C_k \in C$ is scheduled on a unique processor. The resultant schedule can be represented by $S = \{< i, k >, \propto\}$ where $k \in M$ and $i \rightarrow k$ iff $i \in C_k$ (i.e. all nodes i in cluster C_k are scheduled on processor k). The term "$\propto$" represents precedence of execution: for any $i, j \in C_k$, $< i, k > \propto < j, k >$ if $ee(i) \leq ee(j)$. The finish time of execution is T.

Now, the function $F(K)$ is computed in $O(n)$ time

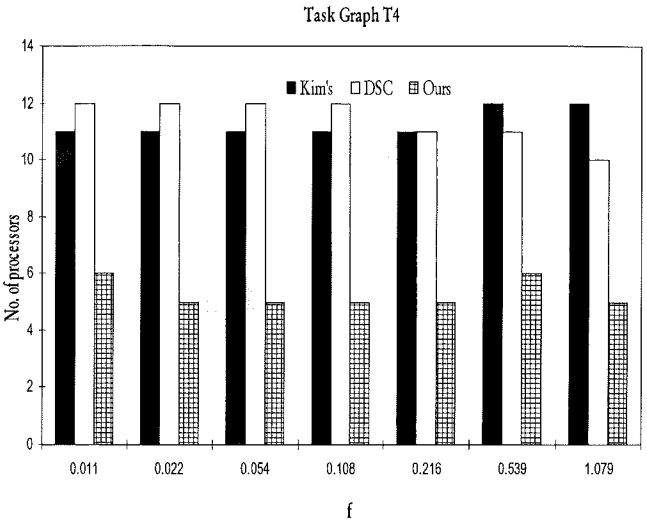


Fig. 16 Comparison of the number of processors needed to obtain the minimum execution times.

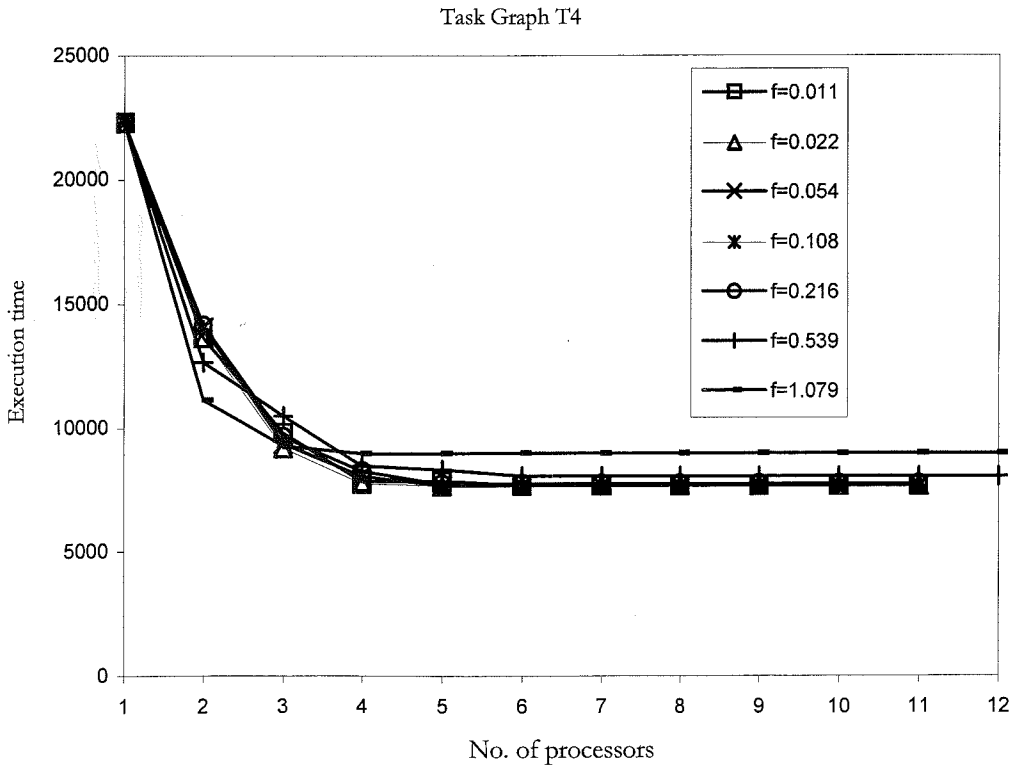


Fig. 15 Execution times using different number of processors.

and so is the function EE when combined with the preceding operation of the union on sets already ordered by earliest event times. Therefore, the worst case time complexity of *MergeClusters* is $O(n^4)$.

4. Performance

Simulations to obtain execution times corresponding to schedules generated by the algorithm *ClusterMerge* and those by the algorithms in [6],[12] were performed on several directed a-cyclic task graphs. In this paper we show results for some representative directed a-cyclic task graphs. These task graphs: T_1, T_2, T_3, T_4, and T_5 are shown in Figs. 2, 6, 10, 14 and 18 respectively. The task graph T_1 is an irregular computation graph and represents a modified molecular dynamics code, the task graph T_2 is a regular computation graph and rep-

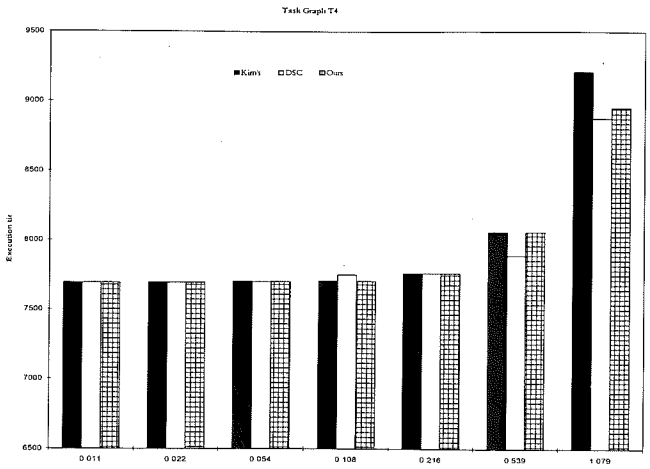


Fig. 17 Comparison of the minimum execution time.

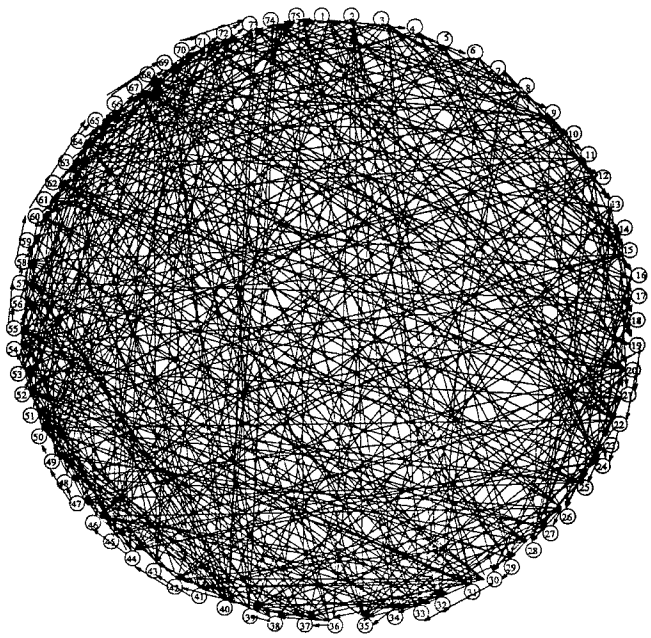


Fig. 18 Task graph T_5.

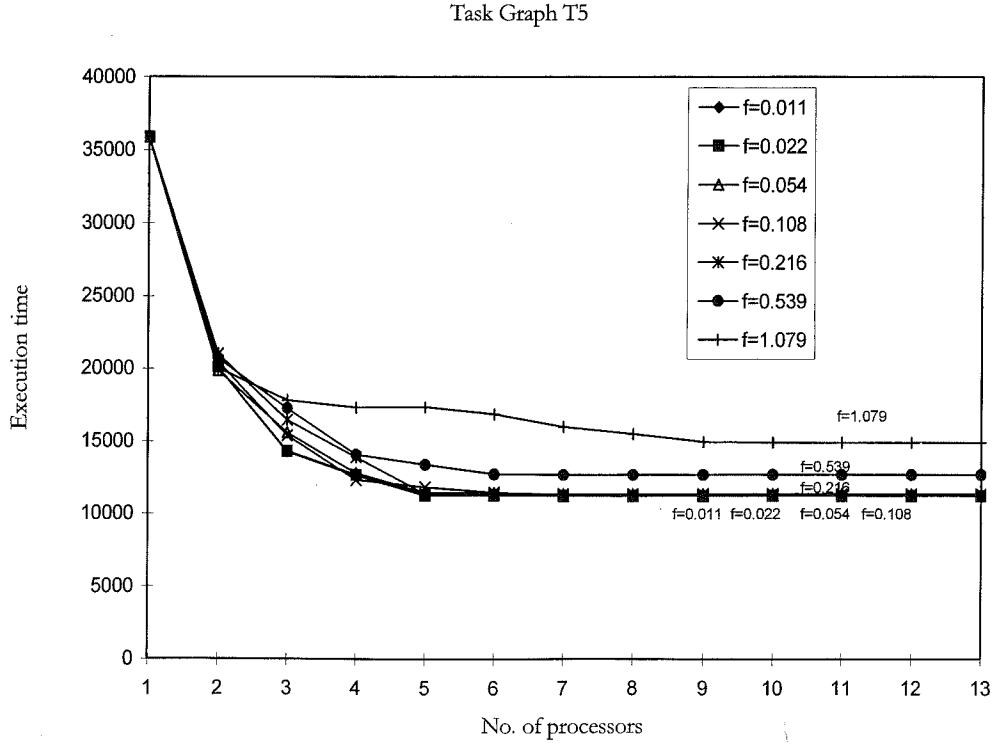


Fig. 19 Execution times using different number of processors.

resents the code for Sieve of Eratosthenes. Both T_1 and T_2 were used in [6]. The task graphs T_3, T_4 and T_5 were randomly obtained. For the task graph T_1, Figs. 3, 4 and 5 show the simulation results. The parameter f in the graphs is the ratio of the average weight of the edges and the average weight of the nodes of the directed a-cyclic task graph. Figure 3 shows the execution times of T_1 when different numbers of processors are employed; the different curves have been obtained by varying f. Figure 4 compares the number of processors used to obtain minimum possible execution times for T_1 using the scheduling algorithms in [6],[12] against algorithm *ClusterMerge* for different values of f. Figure 5 compares the minimum execution times obtained using the scheduling algorithms in [6],[12] against *ClusterMerge* for different values of f. Similar results for task graphs T_2, T_3, T_4 and T_5 are shown in Figs. 7, 8, 9, 11, 12, 13, 15, 16, 17 and 19, 20 and 21 respectively. Similar results were obtained for numerous randomly generated graphs with random node and edge weights.

Let the tuple (f, u, g) represent that for a particular f the number of clusters generated for a task graph by *ClusterNodes* is u and the number of clusters merged by *MergeClusters* without delaying the execution time of the task graph is g. To demonstrate the effectiveness of *MergeClusters* we next list the corresponding tuples for task graphs T_1, T_2, T_3, T_4 and T_5. For task graph T_1 they are: $(0.081, 15, 11)$, $(0.162, 15, 9)$, $(0.405, 15, 9)$, $(0.81, 15, 11)$, $(1.62, 14, 9)$ and $(4.05, 15, 13)$; for task graph T_2 they are: $(0.011, 16, 9)$, $(0.022, 16, 9)$, $(0.055, 16, 9)$, $(0.109, 16, 9)$, $(0.218, 16, 9)$, $(0.546, 16, 9)$ and $(1.092, 16, 10)$; for task graph T_3 they are: $(0.254, 4, 2)$, $(0.51, 4, 2)$, $(1.275, 4, 2)$, $(2.55, 4, 3)$, $(5.08, 4, 3)$, $(7.62, 4, 3)$, $(10.16, 4, 3)$, $(12.71, 4, 3)$, $(15.24, 4, 3)$, $(17.85, 4, 3)$, $(20.4, 4, 3)$, $(22.95, 4, 3)$ and $(25.5, 4, 3)$; for task graph T_4 they are: $(0.011, 11, 5)$, $(0.022, 11, 6)$, $(0.054, 11, 5)$, $(0.108, 11, 5)$, $(0.216, 11, 5)$, $(0.539, 16, 6)$, and $(1.079, 15, 5)$ and for task graph T_5 they are: $(0.01, 12, 7)$, $(0.021, 12, 7)$, $(0.052, 12, 5)$, $(0.104, 12, 5)$, $(0.208, 13, 8)$, $(0.52, 14, 8)$ and $(1.041, 15, 6)$.

From the results it is clear that the algorithm *ClusterMerge* schedules task graphs yielding the same or lower execution times than those obtained using the algorithms in [6],[12] while employing fewer processors. We also observe that even though we might have fewer linear clusters than the number of processors, merging some clusters may give a lower execution time than scheduling each of them on separate processors. In such cases savings in inter-processor communication times outweigh the potential speed up, if any, due to parallel execution. Overall, the minimization of the execution time is restricted by the inherent parallelism in the program and the limitation of any heuristic to fully exploit this parallelism. We hasten to add that *ClusterMerge* will not necessarily yield optimal results in some *rare* cases for the following reason. Merging cluster pairs without looking at combinations in future iterations may not lead to the overall optimum. It is also interesting to note that for many DAGs the biggest drop in execution time occurs with the use of the first four processors. And building completely connected systems with four processors is technically feasible!

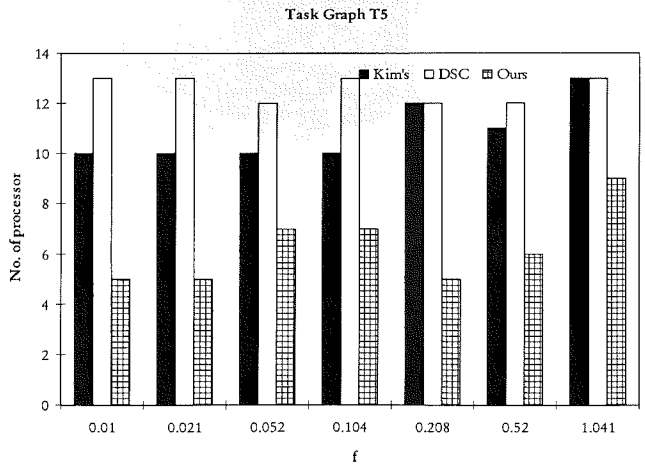


Fig. 20 Comparing the no. of processors needed to obtain the minimum execution times.

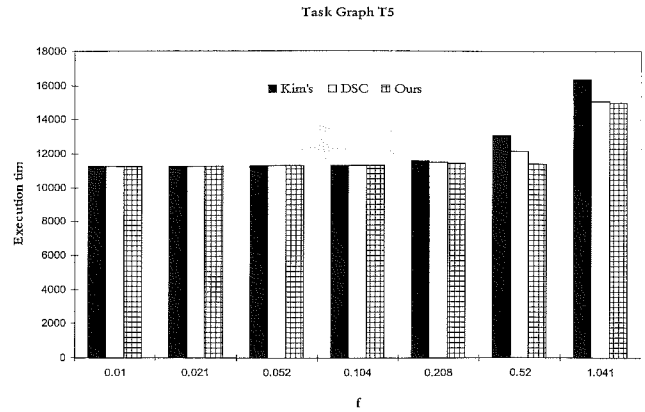


Fig. 21 Comparison of the minimum execution times.

5. Conclusion

We have presented an efficient heuristic algorithm *ClusterMerge* to schedule directed a-cyclic task graphs onto distributed memory multiprocessor systems. We have simulated the performance of the algorithm for numerous task graphs. The results of the simulations clearly demonstrate the superiority of *ClusterMerge*; it yields lower execution times using *fewer* processors than in [6],[12] for *all* task graphs presented in [6] and also on a variety of other task graphs. Furthermore, we have also brought to fore a very important issue: the *impact* of the different interpretations of the longest path in a DAG on scheduling decisions. The algorithm *ClusterMerge* is able to provide a schedule when the user

has only a *certain* number of processors available. Furthermore, we see that for many DAGs the largest percentage drops in execution times occur when using only four processors.

Acknowledgements

The author would like to thank Professor Richard Y. Kain and Dr. Gary Elsesser (Cray Research Inc., SGI) for discussions, reviewing the paper and suggesting several important changes.

References

[1] S. Baskiyar, "Architectural and scheduler support for object-oriented programs," Ph.D. Thesis, University of Minnesota, Minneapolis, 1993.

[2] T.C.E. Cheng and C.C.S. Sin, "A state of the art review of parallel machine scheduling research," European Journal on Operations Research, vol.47, pp.271–292, North-Holland, 1990.

[3] A. Gerasoulis, S. Venugopal, and T. Yang, "Clustering task graphs for message passing architectures," Proc. ACM International Conference on Supercomputing, pp.447–456, 1990.

[4] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of Data Structures in C++, Computer Science Press Inc., 1997.

[5] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," IEEE Trans. Comput., vol.C-33, no.11, Nov. 1984.

[6] S.J. Kim and J.C. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," Proc. International Conference on Parallel Processing, IEEE Computer Society, vol.3, 1988.

[7] E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, "Recent developments in deterministic sequencing and scheduling: A survey," in Deterministic and Stochastic Scheduling, eds. M.A H. Dempster, et al., D. Reidl Publishing Company, 1982.

[8] V. Lo, "Heuristic algorithms for task assignment in distributed systems," IEEE Trans. Comput., vol.37, no.11, 1988.

[9] C. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms," SIAM J. Computing, vol.19, no.2, pp.322–328, 1990.

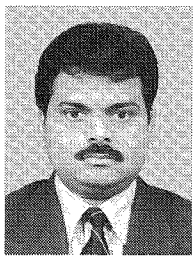
[10] V. Sarkar, Partitioning and Scheduling Parallel Programs for Multiprocessors, The MIT Press, Cambridge, MA, 1989.

[11] J. Ullman, "NP-complete scheduling problems," J. Computer System and Sciences, vol.10, pp.384–393, 1975.

[12] T. Yang and A. Gerasoulis, "DSC: Scheduling parallel tasks on an unbounded number of processors," IEEE Trans. Parallel & Distributed Systems, vol.5, no.9, Sept. 1994.

[13] W. Yu, "LU decomposition on a multiprocessing system with communication delays," Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1984.

[14] R.P. Stanley, Combinatorics and Commutative Algebra, Birkhauser, 1996.



Sanjeev Baskiyar received the B.S. degree in Electronics and Communication Engineering from the Indian Institute of Science, Bangalore and the M.S.E.E. and Ph.D. degrees from the University of Minnesota, Minneapolis. Currently, he is Assistant Professor in the Department of Computer Science and Engineering at Auburn University, Auburn, AL. Previously, he has worked as Assistant Professor at Western Michigan University, as a Senior Software Engineer in the Unisys Corporation, Roseville, MN and as a Computer Engineer in Tata Engineering and Locomotive Company, Jamshedpur. His current research interests are in Task Mapping onto Multiprocessors, Computer Systems Architecture and Real-time Systems.