Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors

Yu-Kwong Kwok, *Member, IEEE Computer Society* and
Ishfaq Ahmad, *Member, IEEE Computer Society*

Abstract—In this paper, we propose a static scheduling algorithm for allocating task graphs to fully connected multiprocessors. We discuss six recently reported scheduling algorithms and show that they possess one drawback or the other which can lead to poor performance. The proposed algorithm, which is called the Dynamic Critical-Path (DCP) scheduling algorithm, is different from the previously proposed algorithms in a number of ways. First, it determines the critical path of the task graph and selects the next node to be scheduled in a dynamic fashion. Second, it rearranges the schedule on each processor dynamically in the sense that the positions of the nodes in the partial schedules are not fixed until all nodes have been considered. Third, it selects a suitable processor for a node by looking ahead the potential start times of the remaining nodes on that processor, and schedules relatively less important nodes to the processors already in use. A global as well as a pair-wise comparison is carried out for all seven algorithms under various scheduling conditions. The DCP algorithm outperforms the previous algorithms by a considerable margin. Despite having a number of new features, the DCP algorithm has admissible time complexity, is economical in terms of the number of processors used and is suitable for a wide range of graph structures.

Index Terms—Algorithms, clustering, list scheduling, multiprocessors, processor allocation, parallel scheduling, task graphs.

1 Introduction

An efficient scheduling of a parallel program onto the processors is vital for achieving a high performance from a parallel computer system. When the structure of the parallel program in terms of its task execution times, task dependencies, task communications and synchronization, is known a priori, scheduling can be accomplished statically at compile time. The objective is to minimize the schedule length. It is well known, however, that multiprocessor scheduling for most precedence-constrained task graphs is an NP-complete problem in its general form [12], [21]. To tackle the problem, simplifying assumptions have been made regarding the task graph structure representing the program and the model for the parallel processor systems [7], [14]. However, the problem is NP-complete even in two simple cases: 1) scheduling unit-time tasks to an arbitrary number of processors [15]; 2) scheduling one or two time unit tasks to two processors [9]. There are only two special cases for which optimal polynomial-time algorithms exist. These cases are: scheduling tree-structured task graphs with identical computation costs on an arbitrary number of processors and scheduling arbitrary task graphs with identical computation costs on two processors [18], [33]. However, even in these cases, no communication is assumed among the tasks of the parallel program. It has been shown that scheduling an arbitrary task graph with intertask communication onto two processors is NP-complete and scheduling a tree-structured task graph with intertask communication onto a system with an arbitrary number of processors is also NP-complete [25].

For more realistic cases, a scheduling algorithm needs to address a number of issues. It should exploit the parallelism by identifying the task graph structure, and take into consideration task granularity, arbitrary computation and communication costs. Moreover, in order to be of practical use, a scheduling algorithm should have low complexity and should be economical in terms of the number of processors used [3], [11]. Because of its vital importance, the scheduling problem continues to be a focus of attention from the research community [4], [5], [8], [13], [16], [17], [19], [20], [22], [23], [24], [27], [28], [29], [30], [31], [34]. In this paper, we propose a new static scheduling algorithm. The proposed algorithm, which is called the Dynamic Critical Path (DCP) algorithm, schedules task graphs with arbitrary computation and communication costs to a multiprocessor system with unlimited number of fully-connected identical processors. The DCP algorithm tackles the drawbacks of previous approaches and outperforms them by a considerable margin. The algorithm has admissible time complexity. It is also economical in terms of the number of processors used and is suitable for different types of graph structures.

The remainder of this paper is organized as follows. In the next section, we describe the background of the scheduling problem including some of the major issues involved. In Section 3, we describe six recently reported scheduling algorithms. The merits and limitations of these algorithms are discussed briefly. In Section 4, we describe our DCP sched-

• *Y.-K. Kwok and I. Ahmad are with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {iahmad, csricky}@cs.ust.hk.*

Manuscript received May 4, 1994; revised Jan. 1, 1995.
For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number D95081.

uling algorithm and discuss its design principles as well as its properties. In Section 5, we use an example to illustrate the functionality of all seven algorithms. In Section 6, we provide the experimental results and a comparison of all algorithms. Section 7 provides the concluding remarks.

2 Background

A parallel program can be represented by a directed acyclic graph $G = (V, E)$, where V is the set of nodes ($|V| = v$) and E is the set of edges ($|E| = e$). A node in the parallel program graph represents a task which is a set of instructions that must be executed serially in the same processor. Associated with each node is its *computation cost*, denoted by $w(n_i)$ which indicates the task execution time. The edges in the parallel program graph correspond to the communication messages and precedence constraints among the nodes. Associated with each edge is a number indicating the time required for communicating the data from one node to another. This number is called the *communication cost* of the edge and is denoted by c_{ij}. Here, the subscript ij indicates that the directed edge emerges from the source node n_i and incidents on the destination node n_j. The source node and the destination node of an edge is called the *parent* node and the *child* node respectively. In a task graph, a node which does not have any parent is called an *entry* node while a node which does not have any child is called an *exit* node. A node cannot start execution before it gathers all of the messages from its parent nodes. The *communication-to-computation ratio* (*CCR*) of a parallel program is defined as its average communication cost divided by its average computation cost on a given system. We assume each processor in the system possesses dedicated hardware to deal with communication so that communication can take place simultaneously with computation. The communication cost among two nodes assigned to the same processor is assumed to be zero.

The objective of static scheduling is to assign the nodes of the task graph to the processors such that the schedule length or *makespan* is minimized without violating the precedence constraints. A schedule is considered *efficient* if the schedule length is short and the number of processors used is reasonable. There are many approaches that can be employed in static scheduling. These include queuing theory, graph theoretic approaches, mathematical programming and state-space search [6], [14]. In the classical approach [1], [9], which is also called *list scheduling*, the basic idea is to make an ordered list of nodes by assigning them some priorities, and then repeatedly execute the following two steps until a valid schedule is obtained.

1) Select from the list the node with the highest priority for scheduling.
2) Select a processor to accommodate this node.

The priorities are determined statically before the scheduling process begins. In the scheduling process, the node with the highest priority is chosen for scheduling. In the second step, the best possible processor, that is, the one which allows the earliest start time, is selected to accommodate this node. Most of the reported scheduling algorithms based on this concept [15], [18], [30] employ variations in the priority assignment methods such as HLF (Highest level First), LP (Longest Path), LPT (Longest Processing Time) and CP (Critical Path).

The main problem with list scheduling algorithms is that static priority assignment may not always order the nodes for scheduling according to their relative importance. A node is more important than other nodes if timely scheduling of the node can lead to a better schedule eventually. The drawback of a static approach is that an inefficient schedule may be generated if a relatively less important node is chosen for scheduling before the more important ones. Static priority assignment may not capture the variation in relative importance of nodes during the scheduling process. For example, consider the task graph shown in Fig. 1 (top left). Here, a schedule is produced using the HLFET (Highest Levels First with Estimated Times) algorithm [1], [18], which determines the priority of a node by computing its *level*. The level of a node is the largest sum of computation costs along a path from the node to an exit node. The node with a higher level gets a higher priority. The HLFET algorithm schedules nodes in the order: n_1, n_2, n_3, n_4. The schedule is shown in Fig. 1 (top right) in which all the nodes are scheduled to one processor (PE denotes a processor); the schedule length is 43 time units. However, the schedule length can be reduced, as shown in Fig. 1 (bottom left), if we schedule the nodes in the order: n_1, n_3, n_2, n_4. At the second scheduling step, n_3 is a relatively more important node than n_2 because if it is not scheduled to start earlier on a processor, the start time of n_4 will be delayed due to the large communication costs along the path $n_1 - n_3 - n_4$. Thus, the HLFET algorithm does not precisely identify the most important node at each scheduling step as it orders nodes by assigning each of them a static attribute which does not depend on the communication among nodes.

As can be seen from the above simple example, a scheduling algorithm may generate very inefficient schedules if it cannot assign accurate priorities to nodes. One important attribute of a task graph that can be used to determine node priorities accurately is explained in the following definition.

Definition 1. *A Critical Path (CP) of a task graph is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation costs and communication costs is the maximum.*

The CP of a task graph potentially determines the schedule length because the cumulative computation costs of the nodes on the CP is the lower bound on the schedule length. Indeed, the final schedule length is the length of the "Critical Path" of the scheduled task graph. For example, the CP of the task graph shown in Fig. 1 (top left) is the path $n_1 - n_3 - n_4$ (shown in thick arrows); while the CP of the scheduled graph shown in Fig. 1 (bottom right) is the path $n_1 - n_2 - n_4$. Thus, generating an efficient schedule requires proper scheduling of the nodes on the CP. We will further elaborate this issue in Section 4.

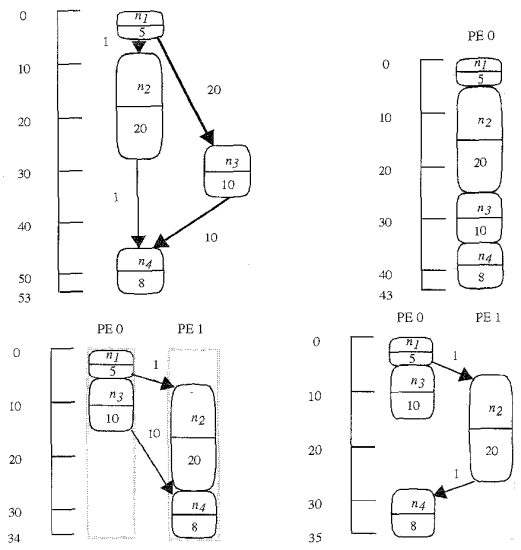


Fig. 1. (Top left) A task graph; (top right) the schedule generated by HLFET, MCP, ETF, and DLS algorithms (schedule length = 43 time units); (bottom left) the schedule generated by the DSC algorithm (schedule length = 34 time units); (bottom right) the schedule generated by the EZ and MD algotithm (schedule length = 35 time units).

In order to avoid scheduling less important nodes before the more important ones, node priorities can be determined dynamically during the scheduling process. The priorities of nodes are recomputed after a node has been scheduled in order to capture the changes in the relative importance of nodes. Thus, the following three steps are repeatedly executed in such kind of scheduling algorithms:

1) Determine new priorities of all unscheduled nodes.
2) Select the node with the highest priority for scheduling.
3) Select the most suitable processor to accommodate this node.

Scheduling algorithms which employ the above three step approach can potentially generate better schedules [13], [35]. However, this can increase the complexity of the algorithm.

3 Related Work

In this section, six recently reported scheduling algorithms and their characteristics are described. These are the Edge-Zeroing (EZ) algorithm [32], the Modified Critical Path (MCP) algorithm [37], the Mobility Directed (MD) algorithm [37], the Earliest Task First (ETF) algorithm [16], the Dynamic Level Scheduling (DLS) algorithm [35], and the Dominant Sequence Clustering (DSC) algorithm [36].

3.1 The EZ Algorithm

As opposed to the CP-based algorithms, the EZ algorithm attempts to reduce the partial schedule length at each step by considering the highest cost edge in the task graph. At each scheduling step, the algorithm schedules the two nodes with the heaviest communication edge to the same processor provided the partial schedule length does not increase. To do this, the EZ algorithm first constructs a list of edges in decreasing order of communication costs. It then removes the first edge from the list and schedules the two incident nodes to the same processor if the partial schedule length is not increased. If the partial schedule length is increased by such scheduling, the two nodes are scheduled to two distinct processors. The nodes within the same processor are maintained in a decreasing order of their *levels* (levels are computed with the same method as used by the HLFET algorithm). The process is repeated until all nodes are scheduled. The complexity of the EZ algorithm is shown to be $O(e\,(e + v))$.

For the task graph shown in Fig. 1 (top left), the EZ algorithm generates the schedule shown in Fig. 1 (bottom right). It is apparent from this example that the criterion used by the EZ algorithm to select node for scheduling does not properly identify the most important node at each scheduling step. For this task graph, the EZ algorithm schedules the nodes in the order: n_1, n_3, n_4, n_2. After n_1 and n_3 are scheduled, the highest cost edge is (n_3, n_4). Thus, n_4 is scheduled to PE 0. However, n_2 cannot be scheduled to PE 0 afterwards in order not to increase the schedule length. This results in an inefficient schedule.

3.2 The MCP Algorithm

The MCP algorithm is designed based on an attribute called the *latest possible start time* of a node. A node's latest possible start time is determined through the *as-late-as-possible* (ALAP) binding, which is done by traversing the task graph upward from the exit nodes to the entry nodes and by pulling the nodes downwards as much as possible constrained by the length of the CP. The MCP algorithm first computes all the latest possible start times for all nodes. Then, each node is associated with a list of latest possible start times which consists of the latest possible start time of the node itself, followed by a decreasing order of the latest possible start times of its children nodes. The MCP algorithm then constructs a list of nodes in an increasing *lexicographical* order of the latest possible start times lists. At each scheduling step, the first node is removed from the list and scheduled to a processor that allows for the earliest start time. The MCP algorithm was originally designed for a bounded number of processors. The complexity of the MCP algorithm is shown to be $O(v^2 \log v)$.

The MCP algorithm assigns higher priorities to nodes which have smaller latest possible start times. However, the MCP algorithm does not necessarily schedule nodes on the CP first. For example, consider the task graph in Fig. 1(top left) again. Here, the MCP algorithm schedules nodes in the same order as the HLFET algorithm and hence generates the same schedule (shown in Fig. 1 (top right)). The MCP algorithm does not assign node priorities accurately even though it takes communication among nodes into account for computing the priorities.

3.3 The MD Algorithm

The MD algorithm selects a node at each step for scheduling based on an attribute called the *relative mobility*. Mobility of a node is defined as the difference between a node's *earliest start time* and latest start time. Similar to the ALAP binding mentioned above, the earliest possible start time is assigned to each node through the *as-soon-as-possible* (ASAP) binding

which is done by traversing the task graph downward from the entry nodes to the exit nodes and by pulling the nodes upward as much as possible. Relative mobility is obtained by dividing the mobility with the node's computation cost. Essentially, a node with zero mobility is a node on the CP. At each step, the MD algorithm schedules the node with the smallest mobility to the first processor which has a large enough time slot to accommodate the node without considering the minimization of the node's start time. After a node is scheduled, all the relative mobilities are updated. The complexity of the MD algorithm is $O(v^3)$.

As opposed to the MCP algorithm, the MD algorithm determines node priorities dynamically. Although the MD algorithm can correctly identify the CP nodes for scheduling at each step, the selection of a suitable time slot and a processor are not done properly. The major problem with the MD algorithm is that it pushes scheduled nodes downwards to create a large enough time slot to accommodate a new node without paying any regard to the degradation in the schedule length. It may happen that pushing down the nodes may increase the final schedule length. The second drawback of the MD algorithm is that it looks for a suitable processor by scanning the processors one by one starting with the first processor. This processor selection criterion does not precisely make any effort to minimize the start time of nodes at each step. Another problem with the MD algorithm is that it inserts a node into an idle time slot on a processor without considering whether the descendants of that node can be scheduled in a timely manner. The schedule generated by the MD algorithm for the task graph in Fig. 1 (top left) is the same as the one generated by the EZ algorithm (shown in Fig. 1 (bottom right)). When node n_4 is considered, it is found that there is a time slot on PE 0 large enough to accommodate it. The MD algorithm schedules n_4 to PE 0 without considering other processors. As a result, a longer schedule length is obtained.

3.4 The ETF Algorithm

Similar to the MCP algorithm, the ETF algorithm uses static node priorities and assumes only a bounded number of processors. However, a node with a higher priority may not necessarily get scheduled before the nodes with lower priorities. This is because at each scheduling step, the ETF algorithm first computes the earliest start times for all the *ready* nodes and then selects the one with the smallest value of the earliest start time. A node is ready if all its parent nodes have been scheduled. The earliest start time of a node is computed by examining the start time of the node on all processors exhaustively. When two nodes have the same value of the earliest start times, the ETF algorithm breaks the tie by scheduling the one with a higher static priority. The static node priorities can be computed based on the node levels as in the HLFET algorithm or the latest possible start times as in the MCP algorithm. The complexity of the ETF algorithm is described to be $O(pv^2)$, where p is the number of processors given.

The major deficiency of the ETF algorithm is that it may not be able to reduce the partial schedule length at every scheduling step. This is because a node which has the smallest value of the earliest start time may not necessarily lie on the CP. An adverse effect of scheduling such nodes before the CP nodes is that the earlier time slots on the processors may be occupied and hence the CP nodes may not get scheduled in a timely manner. It is in this respect that the ETF algorithm works in a similar way as the MCP algorithm. For the task graph shown in Fig. 1(top left), the ETF algorithm generates the same schedule as the MCP algorithm (shown in Fig. 1 (top right)). This is expected because both algorithms attempt to minimize the start time of a node at each step in a greedy fashion.

3.5 The DLS Algorithm

Similar to the MD algorithm, the DLS algorithm determines node priorities dynamically by assigning an attribute called the *dynamic level* (DL) to all unscheduled nodes at each scheduling step. The *DL* is computed by using two quantities. The first quantity is the static level $SL(n_i)$ of a node n_i which is defined as the maximum sum of computation costs along a path from n_i to an exit node. The second quantity is the start time $ST(n_i, J)$ of n_i on processor J. The dynamic level $DL(n_i, J)$ for the node-processor pair (n_i, J) is then defined as $SL(n_i) - ST(n_i, J)$. At each scheduling step, the DLS algorithm computes the *DL* for each ready node on every processors. Then, the node-processor pair which constitutes the largest *DL* among all other pairs is selected so that the node is scheduled to the processor. This process is repeated until all the nodes are scheduled. The complexity of the DLS algorithm is shown to be $O(v^3pf(p))$, where p is the number of processors given and $f(p)$ is the complexity of the data routing algorithm to calculating the *ST* of a node at each step.

The DLS algorithm does not assign priorities based on the CP. It performs exhaustive pair matching[1] of nodes to processors at each step to find the highest priority node. The idea of the DLS algorithm is to use a composite parameter *DL* to select a node with a higher static level and a smaller start time to schedule. However, it should be noted that the level of the selected node may not be the highest and its start time may not be the earliest among all the ready nodes. This is the subtle difference between the DLS algorithm and the ETF algorithm (note that the ETF algorithm tries to schedule a node that can start earlier and breaks ties by using the static levels). At the beginning of the scheduling process, the *DL*s of ready nodes are dominated by the *SL*s because the ready nodes are in higher levels in the task graph; and their start times are likely to be small. On the other hand, when scheduling the nodes in a lower level (say, the exit nodes), the *DL*s of the ready nodes are dominated by their start times on the processors because their *SL*s are small whereas their start times are large. This reveals the flaw in the behavior of the DLS algorithm. A node with a large *SL* may be scheduled first even though its start time is not small. This may block the early scheduling of more important nodes. For the task graph in Fig. 1 (top left), the DLS algorithm generates the schedule shown in Fig. 1 (top right). The DLS algorithm schedules the nodes in the same order as the MCP algorithm and therefore the same schedule is produced.

1. It should be noted that lower complexity versions of the DLS algorithm are reported in [35]. Those versions are streamlined to run faster with degraded performance. However, in our study, we have chosen the version that can give the best performance in terms of schedule lengths.

3.6 The DSC Algorithm

The DSC algorithm is based on an attribute called the *dominant sequence* which is essentially the critical path of the partially scheduled task graph at each step. At each step, the DSC algorithm checks whether the highest CP node is a ready node. If the highest CP node is a ready node, the DSC algorithm schedules it to a processor that allows the minimum start time. Such minimum start time may be achieved by "rescheduling" some of the node's parent nodes to the same processor. On the other hand, if the highest CP node is not a ready node, the DSC algorithm does not select it for scheduling. Instead, the DSC algorithm selects the highest node that lies on a path reaching the CP for scheduling. The DSC algorithm schedules it to the processor that allows the minimum start time of the node provided that such processor selection will not delay the start time of a not yet scheduled CP node. The delayed scheduling of the CP nodes allows the DSC algorithm to incrementally determine the next highest CP node. This strategy also leads to the low complexity of the DSC algorithm.

Although the DSC algorithm can identify the most important node at each scheduling step, it does not schedule a CP node if it is not a ready node. However, delaying the scheduling of a CP node may prevent it from occupying an earlier idle time slot in the subsequent scheduling steps. Another deficiency of the DSC algorithm is that it uses more processors than necessary because it schedules a node to a new processor if its start time cannot be reduced by scheduling to any processor already in use. However, it is possible to save processors by scheduling nodes to processors already in use without degrading the schedule length. The complexity of the DSC algorithm is shown to be $O((e + v) \log v)$. For the task graph in Fig. 1 (top left), the DSC algorithm generates the schedule shown in Fig. 1 (bottom left). The deficiencies mentioned above are not revealed by this example.

4 The Proposed Algorithm

In this section, we present the proposed DCP scheduling algorithm. As discussed earlier, although the six scheduling algorithms described above can produce efficient schedules, each of them has its own deficiencies. The proposed algorithm can overcome the deficiencies of these algorithms and have the following features.

- It assigns dynamic priorities to the nodes at each step based on the *dynamic critical path* (defined below) so that the schedule length can be reduced monotonically.
- It changes the schedule on each processor *dynamically* in that the start times of the nodes are not fixed until all nodes have been scheduled.
- It selects a suitable processor for a node by looking ahead the potential start time of the node's *critical child* node on that processor.
- It does not exhaustively examine all processors for a node. Instead, it only considers the processors that are holding the nodes that communicate with this node.
- It schedules relatively unimportant nodes to the processors already in use in order not to waste processors.

In the following, we discuss some of the principles used in the design of our algorithm. In the first part of the discussion, we describe the techniques used to select a node for scheduling. In the second part, we discuss the criteria used for processor selection. We formalize the DCP scheduling algorithm at the end of this section. Table 1 provides some terms and their meanings that will be used in the subsequent discussion.

TABLE 1
Symbols and Their Meanings

Symbol	Meaning
n_i	The node number of a task in the parallel program task graph
$w(n_i)$	The computation cost of node n_i
c_{ij}	The communication cost of the directed edge from node n_i to n_j
e	The number of edges in the task graph
v	The number of nodes in the task graph
CCR	Communication-to-computation ratio
CP	A critical path of the task graph
DCP	A dynamic critical path of the task graph
$DCPL$	The dynamic critical path length
SL_t	The schedule length at scheduling step t
$PE(n_i)$	The processor which contains node n_i
$AEST(ni, J)$	The absolute earliest possible start time of n_i in processor J
$ALST(ni, J)$	The absolute latest possible start time of n_i in processor J

4.1 Node Selection

As described in Definition 1, during the scheduling process, the Critical Path (CP) of a task graph determines the partial schedule length. Thus, the nodes on the CP have to be scheduled properly in time and space. However, as the scheduling process proceeds, the CP can change dynamically. That is, a node on a CP at one step may not be on the CP at the next step. This is because the communication cost among two nodes is considered zero if the nodes are scheduled to the same processor. In order to distinguish the CP at an intermediate scheduling step from the original CP in the task graph, we call it the *dynamic critical path* (DCP). To reduce the intermediate schedule length monotonically and hence achieve a shorter final schedule length, we need to identify the unscheduled nodes on the DCP. In the following theorem, we formalize the condition for reducing the schedule length monotonically.

THEOREM 1. *Let SL_t be the intermediate schedule length at step t of the scheduling process. If n_i is the highest unscheduled node on the DCP whose start time is minimized at step t, then $SL_{t+1} \leq SL_t$.*

PROOF. Clearly, SL_t is equal to the length of the DCP at step t. If the start time of n_i is minimized, then it cannot be greater than the sum of computation costs and communication costs (with the ones among two nodes scheduled to the same processor taken as zeros) along the DCP from the entry node to n_i. It follows that $SL_{t+1} \leq SL_t$. □

To minimize the final schedule length, we select a node on the DCP for scheduling at each step in the proposed algorithm. In order to identify the nodes on the DCP, we use two attributes for each node: the *lower bound* and *upper bound* on the start time of a node. The computation of the

values of these two attributes is explained in the following definitions. In our approach, the start times of nodes on a processor are not fixed until scheduling completes. Thus, in effect, the nodes are simply "clustered" together in a linear order. The first definition described below gives the lower bound on the start time of a node on a processor.

DEFINITION 2. *The absolute earliest start time of a node n_i in a processor J, denoted by AEST (n_i, J) is recursively defined as follows*:

$$max_{1\le k\le p}\left\{AEST\left(n_{i_k}, PE\left(n_{i_k}\right)\right) + w\left(n_{i_k}\right) + r\left(PE\left(n_{i_k}\right), J\right)c_{i_k i}\right\}$$

*where n_i has p parent nodes and n_{i_k} is the k*th *parent node.* $AEST(n_i, J) = 0$ *if it is an entry node, and* $r\left(PE\left(n_{i_k}\right), J\right) = 1$ *if* $PE\left(n_{i_k}\right) \neq J$ *and zero otherwise.*

According to Definition 2, the *AEST* values can be computed by traversing the task graph in a breadth-first manner beginning from the entry nodes so that when is to be computed, all the *AEST* values of n_i's parent nodes are available. The *AEST* of n_i is then simply the latest data arrival time among all its parent nodes. Note that the above definition captures the condition that the communication among two nodes are taken to be zero if they are in the same processor.

DEFINITION 3. *The dynamic critical path length, denoted by DCPL, is defined as*:

$$max_i\left\{AEST\left(n_i, PE(n_i)\right) + w(n_i)\right\}$$

The value of the *DCPL* is simply the schedule length of the partially scheduled task graph. This is because according to Definition 3, it is computed by taking the maximum value across all the earliest *finish* times. The value of the *DCPL* is useful in that it can be used to determine the upper bound on the start time of a node. This is described in the following definition.

DEFINITION 4. *The absolute latest start time of a node n_i in a processor J, denoted by, ALST (n_i, J) is recursively defined as follows:*

$$min_{1\le m\le q}\left\{ALST\left(n_{i_m}, PE\left(n_{i_m}\right)\right) - r\left(PE\left(n_{i_m}\right), J\right)c_{ii_m} - w(n_i)\right\}$$

*where n_i has q children nodes and n_{i_m} is the m*th *child node.* $ALST(n_i, J) = DCPL - w(n_i)$ *if it is an exit node, and* $r\left(PE\left(n_{i_m}\right), J\right) = 1$ *if* $PE\left(n_{i_m}\right) \neq J$ *and zero otherwise.*

Similar to the computation of the *AEST* values, the values of the *ALST* can also be computed by traversing the task graph in a breadth-first manner but in the reverse direction. Note that the *ALST* values should be computed after the *DCPL* has been computed. With each node assigned *AEST* and *ALST*, the nodes on the *DCP* can be easily identified. In the following theorem, we formalize the condition for a node to be a *DCP* node.

THEOREM 2. *If,* $AEST\left(n_i, PE(n_i)\right) = ALST\left(n_i, PE(n_i)\right)$, *then n_i is a node on the DCP.*

PROOF. Assume on the contrary that n_i is not on the *DCP*. This implies that it does not lie on any path of which the sum of computation costs and communication costs equals *DCPL*. Consider the path with the largest sum of computation costs and communication costs, from an entry node n_p to an exit node n_q, going through n_i. Then, by the definitions of *AEST* and *ALST*, $AEST\left(n_i, PE(n_i)\right)$ is equal to the sum of computation costs and communication costs from n_p to n_i excluding $w(n_i)$; and $ALST\left(n_i, PE(n_i)\right)$ is equal to the sum of computation costs and communication costs from n_i to n_q. This can be deduced from the fact that the path from n_p to n_i is the longest path from any entry node to n_i and the path from n_i to n_q is the longest path from n_i to any exit node. Thus, $AEST\left(n_i, PE(n_i)\right) + ALST\left(n_i, PE(n_i)\right) < DCPL$ which in turn implies that $AEST\left(n_i, PE(n_i)\right) \neq ALST\left(n_i, PE(n_i)\right)$. Thus, n_i is on the DCP. □

Based on Theorem 2, we can identify a *DCP* node simply by checking for equality of its *AEST* and *ALST*. In order to reduce the value of the *DCPL* at each scheduling step, the *DCP* node selected for scheduling is the one that has no unscheduled parent node on the *DCP*. We call this the *highest* node in the *DCP*. This gives a well-defined order of scheduling *DCP* nodes so that each *DCP* node is examined for scheduling after its parent *DCP* node.

4.2 Processor Selection

While we are able to identify a *DCP* node, we still need a method to select an appropriate processor for scheduling that node into the most suitable idle time slot. As discussed earlier, the scheduled nodes are not assigned fixed start times. Rather, they are still assigned values of *AEST* and *ALST*. The only constraint on the scheduled nodes on the same processor is that there is a total order among them which will not be affected by the subsequent scheduling. The unfixed start times of the nodes allow us to insert an important node considered in later steps into an earlier time slot by adjusting the *AEST*s and *ALST*s of the scheduled nodes on a processor. At each step, the algorithm needs to find the most suitable processor which contains the most suitable place in time for a node. We formalize a rule below governing the selection of a valid place in time within a processor for a node.

RULE I. *A node n_i can be inserted into a processor J, which contains the sequence of nodes* $\left\{n_{J_1}, n_{J_2}, \ldots, n_{J_m}\right\}$, *if there exists some "k" such that*

$$min\left\{ALST\left(n_i, J\right) + w\left(n_i\right), ALST\left(N_{J_{k+1}}, J\right)\right\} -$$

$$max\left\{AEST\left(n_i, J\right), AEST\left(n_{J_k}, J\right) + w\left(n_{J_k}\right)\right\} \ge w\left(n_i\right)$$

where, $k = 0, \ldots, m$, $ALST\left(n_{J_{m+1}}, J\right) = \infty$, *and;*

$AEST\left(n_{J_0}, J\right) + w\left(n_{J_0}\right) = 0$ *provided none of the nodes in* $\left\{n_{J_1}, n_{J_2}, \ldots, n_{J_k}\right\}$ *is a descendant node of n_i and none of the nodes in* $\left\{n_{J_{k+1}}, n_{J_2}, \ldots, n_{J_m}\right\}$ *is an ancestor node of n_i.*

The above rule states that n_i can be inserted into a processor if it has a sufficiently large idle time slot, possibly created by delaying the *AESTs* of some nodes, to accommodate it. In order not to violate the precedence constraints among nodes, a node must not be inserted in a time slot before which there is a child node scheduled, or after which there is an ancestor node scheduled. Note that as the only criterion for a node to be a candidate for scheduling is that it is the highest node on the *DCP*, it can happen that not all of its parent nodes have been scheduled.

After n_i is inserted into a processor, the communication costs among the nodes in the processor are set to zero. In addition, to preserve the linearity, a zero cost edge is added from the preceding node to n_i and another zero cost edge is added from n_i to the succeeding node. Thus, n_i's *AEST* and *ALST* can change due to the linear ordering of the nodes according to the start times within the processor. The determination of their new values is explained in the following rule.

RULE II. *If a node n_i is inserted to the processor J, then*

$$AEST(n_i, J) = max\{AEST(n_i, J), AEST(n_{J_l}, J) + w(n_{J_l})\}$$

and

$$ALST(n_i, J) = min\{ALST(n_i, J), + w(n_i), ALST(n_{J_{l+1}}, J)\}$$

where l is a value of k satisfying Rule I.

Using Rule I and Rule II, we can determine the set of processors that can accommodate a node n_i at each step. We can create an idle time slot in a processor by delaying the *AESTs* of the scheduled nodes to accommodate n_i. However, this is not done arbitrarily in our proposed algorithm. When finding an idle time slot in a processor to accommodate a node, in order to minimize the length of the *DCP*, we do not delay the *AESTs* of the scheduled nodes in a processor if possible. That is, we first search if there is already a large enough idle time slot in the processor. This is because delaying the *AESTs* of the scheduled nodes is likely to increase the final schedule length since the final *DCP* may contain previously scheduled nodes. Thus, when we consider inserting a node into a processor, we first find if there is an idle time slot in the processor under the constraint that all nodes are bounded to start at their *AESTs*. If there is no such time slot, we ignore the constraint and find another time slot.

Given a set of candidate processors that can accommodate a node, we need to choose the best processor. As can be noticed from the earlier discussion, the other scheduling algorithms use a very straightforward criterion—selecting the processor which gives the minimum start time for a node. Although Theorem 1 states that the schedule length monotonically decreases if we minimize the start time of nodes in the scheduling process, this strategy can potentially generate very inefficient schedules. For example, it can happen that after a node is scheduled to a processor which gives the earliest start time, its heavily communicated children nodes cannot be scheduled timely on the same processor possibly due to the lack of valid idle time slots. In our proposed algorithm, we do not employ this simple start time minimization strategy. Instead, we use a start time *looking-ahead* strategy, which is given in the following rule.

RULE III. *Suppose that n_i is being considered for scheduling. Let n_c be the child node of n_i which has the smallest difference between its ALST and AEST. Then, n_i should be scheduled to the processor J which gives the smallest value of*

$$AEST(n_i, J) + AEST(n_c, J)$$

where $AEST(n_c, J)$ is computed after tentatively inserting n_i to J.

Using Rule III, a node may not be inserted into a processor which allows its earliest start time in the scheduling process. This happens when it is found that the start times of its children nodes are large. Thus, by using this looking-ahead strategy for examining the start times of critical children, the proposed algorithm can avoid scheduling a node to an inappropriate processor. As a result, it avoids the danger of increasing the schedule length in subsequent steps.

Exhaustively examining all the processors to select a suitable one can be very time consuming when the task graph is very large (hence the number of processors to be examined is also large). Observe that the start time of a node can only be reduced by scheduling it to a processor which holds its parent nodes. And, in order to reduce the start times of the node's earlier scheduled children nodes, the processors holding such children nodes are also candidates for examination. Thus, the set of processors to be examined can be restricted to those holding the parent nodes and possibly children nodes, together with a new processor.

It should be noted that at some scheduling step, there may not be any unscheduled node with equal values of *AEST* and *ALST*. This implies that the *DCP* contains only the scheduled nodes and will not change in the subsequent scheduling steps. Thus, in the subsequent scheduling steps, there is no need to delay the *AESTs* of the scheduled nodes into a processor when considering to insert an unscheduled node. This is because making such a node to start earlier will not improve the final schedule length. Consequently, we can schedule each non-*DCP* node to any processor which can accommodate it without increasing the *DCPL*. That is, we can insert the non-*DCP* nodes to any processor provided the schedule length is not increased.

4.3 The DCP Algorithm

The *DCP* algorithm is formalized in this section. It uses two procedures: *Find_Slot*( ) and *Select_Processor*( ) which are described below.

4.3.1 Find_Slot(n_i, J, Condition)

1) Determine $AEST(n_i, J)$ and $ALST(n_i, J)$ on J by taking all communication costs among n_i and its parent nodes and children nodes scheduled on J to be zero
2) If *Condition* = *DONT_PUSH*, then check if there exists k satisfying Rule I without delaying the *AEST* of any node in processor J; otherwise, check if there exists k satisfying Rule I possibly by delaying the *AESTs* of some nodes in processor J
3) **return** $\max\{AEST(n_i, J), AEST(n_{J_l}, J) + w(n_{J_l})\}$ if there exists such k with l being the smallest one; otherwise **return** ∞.

The procedure *Find_Slot*( ) checks whether there is a valid time slot in the processor J to accommodate n_i by using Rule I. In addition to a node and a processor number, *Find_Slot*( ) takes a Boolean parameter *Condition* which indicates if delaying of *AEST*s of scheduled nodes in the processor J is allowed. The complexity of *Find_Slot*( ) is $O(v)$ since there are $O(v)$ nodes in a processor to be examined in order to find a valid time slot in the worst case.

4.3.2 Select_Processor(n_i, Location)

1) If *Location* = *On_DCP*, then construct *Processor_List* in the order: processors holding the parent nodes of n_i, processors holding the children nodes of n_i and a new processor; otherwise, construct *Processor_List* by only including all the processors already in use
2) *Best_Processor* ← NULL
3) *Best_Composite_AEST* ←∞
4) **while** *Processor_List* is not empty **do**
5) J ← remove the first processor from *Processor_List*
6) *This_AEST* ← *Find_Slot*(n_i, J, *DONT_PUSH*)
7) **if** *This_AEST* = ∞ **and** *Location* = *On_DCP* **then**
8) *This_AEST* ← *Find_Slot*(n_i, J, *PUSH*)
9) **end if**
10) **if** *This_AEST* ≠∞ **then**
11) n_c ← unscheduled child node of n_i with the smallest difference between its *ALST* and *AEST*
12) Tentatively insert n_i into J
13) *Child_AEST* ← *Find_Slot*(n_c, J, *DONT_PUSH*)
14) **if** *Child_AEST* + *This_AEST* < *Best_Composite_AEST* **then**
15) *Best_Processor* ← J
16) *Best_Composite_AEST* ← *Child_AEST* + *This_AEST*
17) **end if**
18) **end if**
19) **end while**
20) Schedule n_i to Best_Processor. If n_i cannot get scheduled (n_i is not a DCP node), schedule it to a new processor.

The procedure *Select_Processor*( ) first constructs a processors list in order to find the most suitable one to accommodate n_i. If n_i is on the *DCP*, only the processors containing its parent nodes and children nodes are considered because only these processors can possibly satisfy Rule III as discussed earlier. The processors containing the parent nodes of n_i are given higher priorities to accommodate n_i. This will help reducing the start times of other descendant nodes of n_i that are examined later. If n_i is not on the *DCP*, *Select_Processor*( ) tries all the processors already in use because a processor which does not contain any of its parent nodes and children nodes is also suitable, provided that it has a large enough time slot. However, no delaying is allowed for scheduling such a node since it is undesirable to increase the start times of the scheduled nodes. In selecting the most suitable processor, a parameter composed of the node's *AEST* and also its critical child node's *AEST* is used. Using this parameter can avoid scheduling the node too early to a processor that cannot accommodate the node's critical child node. The complexity of *Select_Processor*( ) is $O(v^2)$. With the procedures *Find_Slot*( ) and *Select_Processor*( ), the *DCP* algorithm is formalized below.

4.3.3 The DCP Algorithm

1) Compute *AEST* and *ALST* for all nodes
2) **while not** all nodes scheduled **do**
3) n_i ← the highest node with the smallest difference between its *ALST* and *AEST*; break ties by choosing the one with a smaller *AEST*
4) If n_i's *ALST* is equal to its *AEST*, then call *Select_Processor*(n_i, *On_DCP*); otherwise, call *Select_Processor*(n_i, *Not_On_DCP*)
5) Update *AEST* and *ALST* for all nodes
6) **end while**
7) Make all nodes' start times to be their respective *AEST*s

The *DCP* algorithm continues to perform scheduling all the *DCP* nodes first. It updates the *AEST* and *ALST* values dynamically after each scheduling step in order to determine the next *DCP* node. Finally, it assigns the actual start times of each node to be its *AEST*. The complexity of the algorithm is $O(v^3)$ because there is v calls to the procedure *Select_Processor*( ).

5 An Application Example

In this section, an example task graph is used to illustrate the effectiveness of the proposed algorithm. For comparison, the schedules generated by the other six scheduling algorithms discussed earlier are also presented. The task graph used is a macro data-flow graph which represents the parallel Gaussian elimination algorithm written in an SPMD style [10], [37] and is shown in Fig. 2. Note that the edges in the two CPs in this task graph are shown with thick arrows.

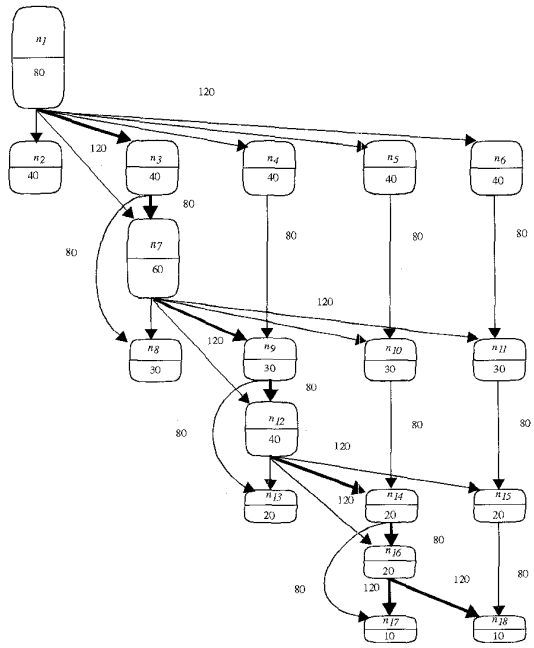


Fig. 2. A parallel Gaussian elimination task graph.

The schedule of the Gaussian elimination task graph generated by the EZ algorithm is shown in Fig. 3(left). The EZ algorithm, as mentioned above, creates a list of edges sorted in a descending order of communication costs. According to the edge list for this example, it schedules nodes in the order: n_1, n_7, n_6, n_5, n_4, n_3, n_2, n_{12}, n_{11}, n_{10}, n_9, n_8, n_{16}, n_{15}, n_{14}, n_{13}, n_{18}, n_{17}. Note that the nodes scheduled to the processor are ordered in decreasing static levels. Thus, for example, although n_2 is selected for scheduling earlier than n_{12}, it is scheduled as the last node in the processor eventually because it has the smallest static level (which is equal to its computation cost). It can be seen that most of the CP nodes are not scheduled to occupy earlier time slots. Instead, the relatively less important nodes, such as n_5, n_6, are scheduled to occupy the important time slots. The task graph in this example is "over-clustered" by the EZ algorithm. The EZ algorithm has the tendency of packing tasks together by reducing parallelism. This is because the EZ algorithm assigns higher priorities to the nodes of the edge having the highest communication cost. These nodes, which may not necessarily be on the CP, may then be scheduled to occupy earlier time slots. It should be noted that the final schedule length may not be reduced by such edge-zeroing action. When the more important nodes are considered at later steps, they may have to be scheduled within later idle time slots. This effect propagates downward along the CP and can eventually lead to a longer schedule length.

The schedule of the Gaussian elimination task graph generated by the MCP algorithm is shown in Fig. 3(right). The MCP algorithm schedules the task graph in the order: n_1, n_3, n_7, n_4, n_9, n_5, n_{12}, n_{10}, n_6, n_{14}, n_{11}, n_{16}, n_{15}, n_2, n_8, n_{13}, n_{17}, n_{18}. The MCP algorithm schedules nodes properly until it considers node n_{10}. The nodes n_1 to n_{12} are scheduled to start at the earliest possible times. However, when n_{10} is considered, it is found that its start time on PE 1 is 300 while its start time on PE 0 is 320. Thus, n_{10} is scheduled to PE 1. Consider the scheduling of the node n_{14}. Since n_{10} has been scheduled to PE 1, n_{14} can start only at time 410 on either PE 0 or PE 1 because of the data dependency from nodes n_{10} and n_{12}. The MCP algorithm schedules it to PE 0 since it selects processors from left to right. If n_{10} were scheduled to PE 0 instead of PE 1, n_{14} could have started earlier. Thus, scheduling n_{10} to PE 1 is not an intelligent decision because this delays the start time of n_{10}s descendants. Similar to n_{10}, the node n_{11} is scheduled to start at its earliest possible time on PE 2. Similarly, n_{16} is scheduled to PE 0 which gives the smallest start time (as n_{14} is scheduled to PE 0). Notice that the adverse effect of inappropriate scheduling of n_{10} propagates downward. In subsequent steps, n_{15} is scheduled on PE 1, to which n_{11} has been scheduled. The nodes n_2, n_8, n_{13} and n_{17} are scheduled to start at the earliest times. However, it should be noted that n_{17} can start earlier if n_{16} is properly scheduled. Similar to the case of n_{10}, the improper scheduling of n_{15} affects the scheduling of n_{18}. It is apparent from this example that the MCP algorithm does not take care of the scheduling of the descendants of a node due to the straightforward start time minimization greedy strategy.

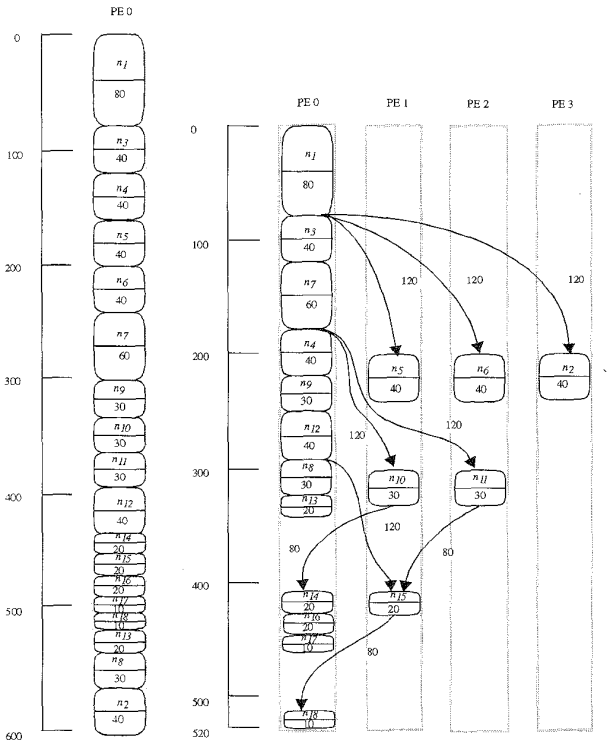


Fig. 3. The schedule of the Gaussian elimination taskgraph generated by (left) the EZ algorithm (schedule length = 600 time units); (right) the ETF, DLS, and MCP algorithms (schedule length = 520 time units).

The DLS algorithm schedules nodes in the order: n_1, n_3, n_7, n_4, n_5, n_6, n_9, n_2, n_{12}, n_{10}, n_{11}, n_8, n_{13}, n_{14}, n_{15}, n_{16}, n_{17}, n_{18} and generates the same schedule as the MCP algorithm. Similar to the MCP algorithm, the DLS algorithm schedules nodes properly until it considers n_{10}. The critical nodes, such as n_{10} and n_{15} are also not scheduled properly. As discussed above, the DLS algorithm schedules a node to a processor which gives the minimum start time without paying any regard to handling the descendant nodes. In this respect, the DLS algorithm has the same problems as the MCP algorithm, as shown in this example. The ETF algorithm also generates the same schedule as the MCP algorithm with the following order: n_1, n_3, n_7, n_4, n_5, n_6, n_2, n_9, n_{12}, n_8, n_{10}, n_{11}, n_{13}, n_{14}, n_{15}, n_{16}, n_{17}, n_{18}. The ETF algorithm selects nodes for scheduling based on start times only. For example, n_2 is selected for scheduling before n_9 because it has a smaller start time.

The schedule generated by the DSC algorithm is shown in Fig. 4. The schedule length is shorter than those of the MCP, DLS, and ETF algorithms. The DSC algorithm schedules the nodes in the order: n_1 n_3, n_7, n_4, n_9, n_{12}, n_{13}, n_8, n_6, n_5, n_{10}, n_{14}, n_{16}, n_{17}, n_{11}, n_{15}, n_{18}, n_2. Similar to the MCP and DLS algorithms, n_{10} is not properly scheduled because of the same reason—the DSC algorithm tries to minimize the start time of a node at each step without considering the effect on subsequent scheduling of the descendant nodes. As can be seen, n_{14} and n_{16} cannot start earlier due to the inappropriate scheduling of n_{10}. When n_{15} is considered, it is scheduled to start at time 410 on PE 5. However, when n_{18} is considered, it is found that its start

time can be reduced from 550 (on PE 4) to 480 (also on PE 4) by rescheduling n_{15} to PE 0 from PE 5. This rescheduling process causes the DSC algorithm to generate a better schedule than the MCP, DLS, and ETF algorithms. Note that this process is not applicable to the scheduling of n_{14} because the DSC algorithm finds that n_{14}'s start time cannot be reduced even if n_{10} is rescheduled. It can be seen that the set of nodes $\{n_{14}, n_{15}, n_{16}, n_{17}, n_{18}\}$ is scheduled to PE 0 but these nodes still cannot start at the earliest times. This is because after n_{10} has to wait for the data from n_5. Although n_5 can start at its earliest time, the schedule length cannot be improved. Similar to the MCP, DLS, and ETF algorithms, the straightforward start time minimization strategy of the DSC algorithm also makes it unable to reduce the start times of the CP nodes $\{n_{14}, n_{16}, n_{17}, n_{18}\}$ by delaying the start times of nodes n_5 and n_{10} even though it can avoid the problem of scheduling node n_{15} too early. Furthermore, the DSC algorithm wastes processors without improving the schedule length.

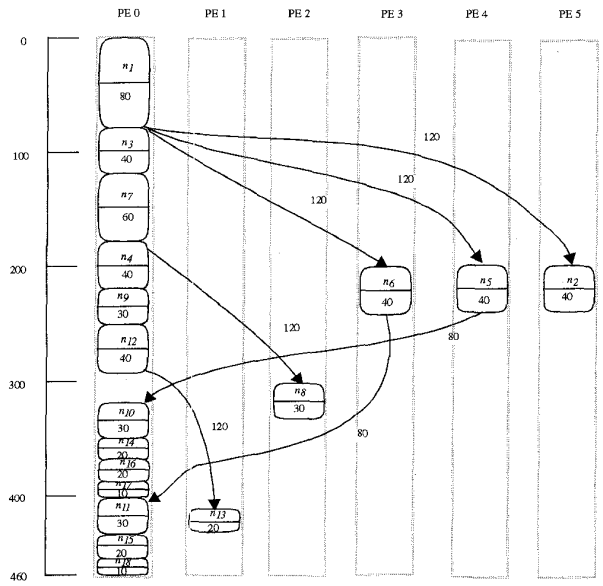


Fig. 4. The schedule of the Gaussian elimination task graph generated by the DSC algorithm (schedule length = 460 time units).

The schedule generated by the MD algorithm is shown in Fig. 5(left). The MD algorithm generates a better schedule compared to the above five algorithms. The MD algorithm schedules nodes in the order: n_1, n_3, n_7, n_4, n_9, n_{12}, n_5, n_{10}, n_{14}, n_{16}, n_6, n_{11}, n_{15}, n_{18}, n_{17}, n_{13}, n_8, n_2. As can be seen, all the CP nodes are scheduled to the same processor PE 0. The MD algorithm is able to avoid scheduling nodes n_{10} and n_{15} too early so that the set of nodes $\{n_{14}, n_{16}, n_{17}, n_{18}\}$ can start immediately after the previous one finishes. However, it has one major problem which makes the schedule length still longer than the best possible. As mentioned in earlier discussion, the MD algorithm scans for a suitable processor for a node from left to right. It schedules the node to a processor which has a large enough idle time slot to accommodate that node without making any effort to minimize the start time. Notice that n_{10} is scheduled to PE 0 instead of PE 1 because the MD algorithm scans for suitable processor from left to right. Thus, n_{10} is "accidentally" scheduled to occupy a proper idle time slot. Consequently, nodes n_{14} and n_{16} can start at their earliest possible times. Consider, for example, the scheduling of n_8. As there is a large enough slot, which is created by pushing nodes downward, on PE 0 to accommodate n_8, it is scheduled to PE 0. Obviously, n_8 is a relatively unimportant node compared with the nodes n_4, n_9, n_{12}, n_{10}, n_{14} and n_{16}. Pushing these nodes downward leads to an inefficient schedule. The MD algorithm also accidentally schedules n_{15} to the proper processor so that n_{18} can also be properly scheduled.

The schedule generated by the DCP algorithm is shown in Fig. 5(right). Let us examine the scheduling process of the DCP algorithm step by step. The *AEST* and *ALST* values of all nodes are shown in Table 2a. As can be noticed, the nodes on the CP can be identified by those having equal values of *AEST* and *ALST*. At this step, the highest CP node n_1 is selected for scheduling. After scheduling the first three CP nodes $\{n_1, n_3, n_7\}$, the *AEST* and *ALST* values of all nodes are shown in Table 2b. The scheduled nodes are marked by asterisks. From this table, we can observe that the CP changes to become $\{n_1, n_4, n_9, n_{12}, n_{14}, n_{16}, n_{17}\}$ (note that the last node can also be n_{18}). The DCP algorithm then selects n_4 to be the next node for scheduling. It is apparent from this scenario that using the *AEST* and *ALST* values, the DCP algorithm can always select the most important node for scheduling. The scheduling steps of the DCP algorithm are depicted in Table 3. In the table, we show the node selected for scheduling as well as its critical child at each step. There are also four columns showing the "composite *AEST*" values with the first number of each entry being the *AEST* of the node to be scheduled, whereas the second number being the *AEST* of its critical child. Although the DCP algorithm assumes the availability of unlimited number of processors, only one new processor is considered at each step. This is because the *AEST* value of a node cannot be improved even if more new processors are considered. One related point is that the schedule length cannot be improved whenever a node is scheduled to a new processor. This is obvious because no communication cost of the edges to the node can be zeroed. Also note that only the processors holding the parent and children nodes of a node are considered. Thus, for example, PE 1 is not considered for scheduling of the node n_{14}. Similarly, PE 1 and PE 2 are not considered for scheduling of the node n_{16}. To see how the "looking ahead" processor selection strategy works, consider the scheduling of the node n_{10}. As can be noticed, its *AEST* value is the smallest if it is scheduled to PE 1. However, since its critical child node n_{14} has a much larger *AEST* value on PE 1 than on PE 0, the node n_{10} is consequently scheduled to PE 0 instead of PE 1. Finally, it should be noted that after the node n_{18} is scheduled, all the CP nodes have been scheduled and the schedule length cannot be improved any further. And, the DCP algorithm only examines the three processors already in use (i.e., PE 0, PE 1 and PE 2) for the scheduling of the remaining three relatively unimportant nodes n_{13}, n_8 and n_2 (if this is not done, the node n_2 will be scheduled to a new processor which can allow a smaller *AEST* value). Eventually, a schedule length of 440 time units is obtained.

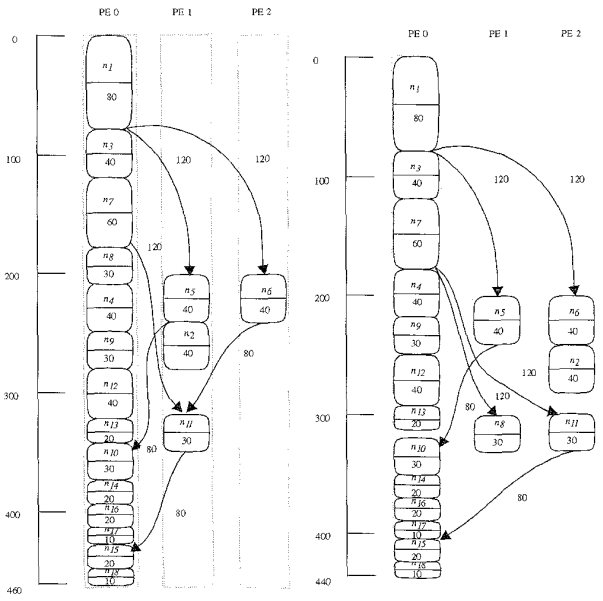


Fig. 5. The schedules of the Gaussian elimination task graph generated by (left) the MD algorithm (schedule length = 460 time units); (right) the DCP algorithm (schedule length = 440 time units).

TABLE 2
THE *AEST* AND *ALST* VALUES OF THE NODES IN THE GAUSSIAN ELIMINATION TASK GRAPH

(a)

Node	*AEST*	*ALST*
n_1 (CP)	0	0
n_2	200	980
n_3 (CP)	200	200
n_4	200	380
n_5	200	540
n_6	200	680
n_7 (CP)	320	320
n_8	500	990
n_9 (CP)	500	500
n_{10}	500	660
n_{11}	500	800
n_{12} (CP)	610	610
n_{13}	770	1000
n_{14} (CP)	770	770
n_{15}	770	910
n_{16} (CP)	870	870
n_{17} (CP)	1010	1010
n_{18} (CP)	1010	1010

(b)

Node	*AEST*	*ALST*
$^*n_1$ (CP)	0	0
n_2	200	800
$^*n_3$	80	100
n_4 (CP)	200	200
n_5	200	360
n_6	200	500
$^*n_7$	120	140
n_8	300	810
n_9 (CP)	320	320
n_{10}	320	480
n_{11}	320	620
n_{12} (CP)	430	430
n_{13}	590	820
n_{14} (CP)	590	590
n_{15}	590	730
n_{16} (CP)	690	690
n_{17} (CP)	830	830
n_{18} (CP)	830	830

(a) before scheduling the first node,
(b) after scheduling three nodes.

6 PERFORMANCE AND COMPARISON

In this section, we present a performance comparison of all seven algorithms. For this purpose, we consider a large set of task graphs as the workload for testing the algorithms. The set contains regular task graphs representing various parallel algorithms and also synthetic task graphs representing commonly encountered algorithmic structures. The generation of the workload is described in the first subsection. The performance comparison is carried out in four contexts. First, we compare the schedule lengths generated by the algorithms. Second, we present a global pair-wise comparison of all algorithms so that we can rank the algorithms by their performance. Third, we compare the number of processors used by the algorithms. Finally, we compare the average running times of these algorithms on a Sun SPARC IPX workstation.

TABLE 3
THE SCHEDULING STEPS OF THE DCP ALGORITHM FOR THE TASK GRAPH IN FIGURE 2

Step	Node	Critical Child	Composite *AEST* PE0	PE1	PE2	PE3	Sch. to	SL
1	n_1	n_7	0 + 320	N.C.	N.C.	N.C.	PE0	1020
2	n_3	n_7	80 + 120	200 + 240	N.C.	N.C.	PE0	900
3	n_7	n_{12}	120 + 490	200 + 490	N.C.	N.C.	PE0	840
4	n_4	n_9	180 + 220	200 + 300	N.C.	N.C.	PE0	820
5	n_9	n_{12}	220 + 250	300 + 330	N.C.	N.C.	PE0	740
6	n_{12}	n_{16}	250 + 590	330 + 590	N.C.	N.C.	PE0	680
7	n_5	n_{10}	N.R.	200 + 300	N.C.	N.C.	PE1	680
8	n_{10}	n_{14}	320 + 350	300 + 410	320 + 410	N.C.	PE0	680
9	n_{14}	n_{17}	350 + 670	N.C.	430 + 670	N.C.	PE0	600
10	n_{16}	n_{18}	370 + 530	N.C.	450 + 530	N.C.	PE0	540
11	n_6	n_{11}	N.R.	N.C.	200 + 300	N.C.	PE2	540
12	n_{11}	n_{15}	N.C.	N.C.	300 + 410	320 + 410	PE2	520
13	n_{17}	NIL	390 + 0	N.C.	N.C.	510 + 0	PE0	520
14	n_{15}	n_{18}	410 + 430	N.C.	410 + 510	410 + 510	PE0	520
15	n_{18}	NIL	430 + 0	N.C.	N.C.	510 + 0	PE0	440
16	n_{13}	NIL	290 + 0	410 + 0	410 + 0	N.C.	PE0	440
17	n_8	NIL	N.R.	300 + 0	330 + 0	N.C.	PE1	440
18	n_2	NIL	N.R.	240 + 0	240 + 0	N.C.	PE2	440

(SL denotes schedule length; "N.R." indicates there is "no room" for the node on the processor; "N.C." indicates the processor is "not considered")

6.1 Workload

In our study, we first considered the macro data-flow graphs for the Gaussian elimination algorithm of different sizes. We also considered the macro data-flow graphs for three other parallel algorithms: fast Fourier transform (FFT) [2], mean value analysis LU-decomposition [26], and Laplace equation solver [37]. These task graphs correspond to the macro data-flow graphs for the corresponding parallel algorithms written in a SPMD style for distributed-memory systems. In addition, we generated synthetic task graphs of various commonly encountered structures: in-tree, out-tree, fork-join, and completely random task graphs [2]. For each category, we generated a number of graphs by varying the number of nodes and values of CCR. For the in-tree, out-tree and fork-join task graphs, edges between two levels were randomly placed. The cost of each node was randomly selected from a normal distribution with mean equal to the specified average computation cost. The cost of each edge was also randomly selected from a normal distribution with mean equal to the product of the average computation cost and the CCR. Miniature examples of each type of graph are shown in Fig. 6.

The Gaussian elimination, LU-decomposition, mean value analysis, and Laplace equation solver algorithms can be characterized by the size of the input data matrix because the number of nodes and edges in the task graph depends on the size of this matrix. For example, the task graph for Gaussian elimination algorithm shown in Fig. 2 is for a matrix of size 4. The number of nodes in the task graphs of these algorithms is roughly $O(N^2)$ where N is the size of the matrix. For our experiments, we varied the matrix sizes so that the graph size ranged from about 20 to 200 nodes. For FFT task graphs, the graph size is roughly $O(M2^M)$ where M is the number of input points which is called the order of the FFT. Again, we varied the orders so that

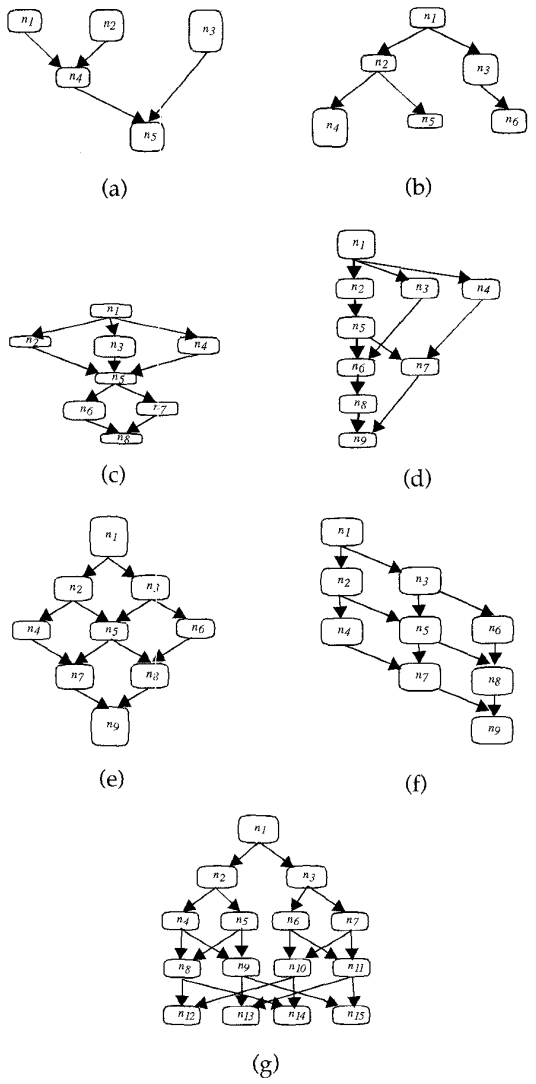


Fig. 6. Miniature example for (a) an in-tree task graph; (b) an out-tree task graph; (c) a fork-join task graph; (d) a LU-decomposition task graph; (e) a mean value analysis task graph; (f) a Laplace equation solver task graph; and (g) a FFT task graph.

the graph size ranged from about 20 to 200 nodes. For the in-tree, out-tree, fork-join, and completely random task graphs, we varied the number of nodes from 20 to 200 with increments of 20. For each size of the task graph, we generated six different graphs for CCR equal to 0.1, 0.5, 1.0, 2.0, 5.0 and 10.0.

6.2 Comparison of Schedule Lengths

For the first comparison, we present the schedule lengths produced by each algorithm for various types of task graphs. The normalized schedule lengths (NSL) for each type of graph structure are given in the charts shown in Fig. 7 to Fig. 15, which were obtained by normalizing the schedule lengths produced by each algorithms to the lower bound. This lower bound was determined by taking the sum of computation costs of the nodes on the original critical path. It should be noted that the lower bound may not always be possible to achieve, and the optimal schedule length may be greater than this bound. Each of these figures also contains a ranking of the algorithms based on the observed schedule lengths for that particular task graph structure. This ranking indicates how each algorithm performed for that type of task graphs.

As can be observed from Fig. 7 to Fig. 15, the ranking of the DLS, MCP, MD, ETF, DSC, and EZ algorithms varies from graph to graph while the DCP algorithm ranks as the best for all types of graphs. From these charts, we also observe that the values of the NSL for all algorithms show a slightly increasing trend if the task graph size is increased. This is due to the fact that the proportion of nodes other than those on the CP increases which makes it difficult to reach the lower bound. In the following, we discuss the relative performance of all the algorithms for each type of task graph.

- *Gaussian elimination graphs.* For the Gaussian elimination task graphs, the CP-based algorithms (DCP, MD, MCP, and DSC) have better performance (see Fig. 7). This is because each Gaussian elimination task graph has only one or two dominating CPs. Thus, an efficient scheduling of nodes on the CPs can lead to good schedules. However, we found that the re-scheduling process in DSC can lead to generating very inefficient schedules, especially when the task graph is large. As EZ only attempts to reduce the communication instead of fully exploiting the parallelism, it also generates very inefficient schedules.
- *Laplace equation solver graphs.* For the Laplace equation solver task graphs (see Fig. 8), MD, MCP and DSC did not show good performance because these algorithms cannot efficiently schedule nodes on the CPs. The reason is that there are many intervening CPs in every Laplace equation solver task graph. It can be seen that DCP outperformed other algorithms by a very large margin. This is because DCP can exploit the inherent parallelism by not scheduling some of the CP nodes too early in the Laplace equation task graphs by the looking-ahead strategy. DSC performed better than both MD and MCP because it schedules CP nodes as soon as possible by making use of more processors. On the other hand, ETF and DLS produced a similar performance.
- *LU-decomposition graphs.* As can be noticed from Fig. 9, in the case of LU-decomposition, DCP again performed the best out of all other algorithms. The ranking of other algorithms is almost the same as in the case of Gaussian elimination task graph probably because the LU-decomposition task graph also has only one CP. We can observe that DLS and MCP yielded a similar performance. Also, DSC, MD and ETF gave a similar performance that is slightly worse than DLS and MCP.
- *FFT graphs.* For the FFT task graphs, we can see from Fig. 10 that the CP-based algorithms performed slightly better. This is due to the fact that although all paths in a FFT task graph are CPs, there is not much intervention among them. DCP and MD performed slightly better than the other CP-based algorithms probably because both of them do not delay the scheduling of the CP nodes.

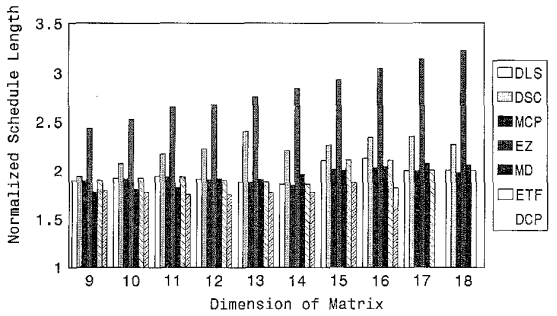


Fig. 7. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for Gaussian elimination graph; algorithm ranking; DCP, (MD, MCP, ETF), DLS, DSC, EZ.

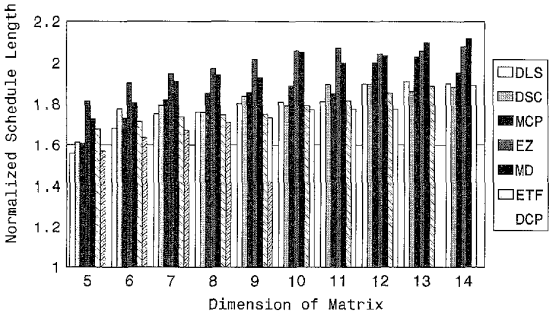


Fig. 8. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for Laplace equation graph; algorithm ranking: DCP, DLS, ETF, DSC, MCP, MD, EZ.

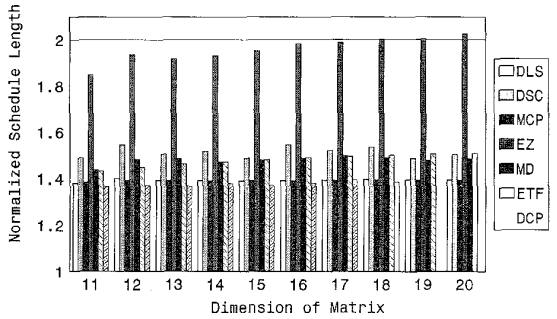


Fig.9. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for LU-Decomposition graph; algorithm ranking; DCP, MCP, DLS, MD, ETF, DSC, EZ.

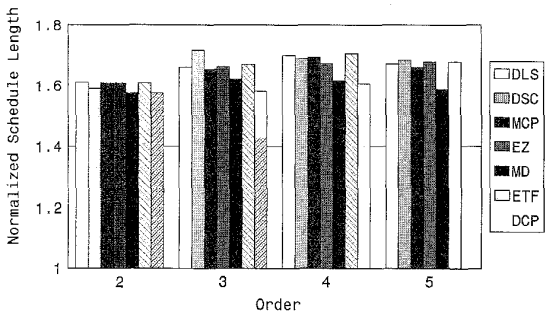


Fig. 10.Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for fast Fourier transform graph; algorithm ranking; DCP, MD, MCP, ETF, DLS, EZ, DSC.

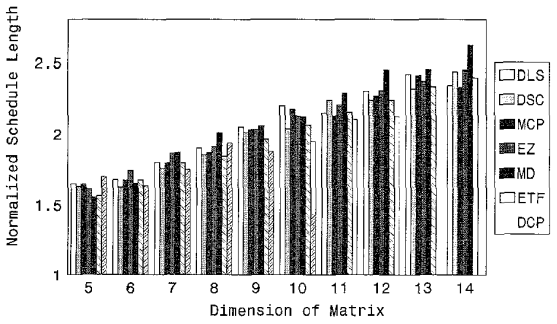


Fig. 11. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for mean value analysis graph; algorithm ranking; DCP, (ETF, DSC, MCP), (DLS, EZ), MD.

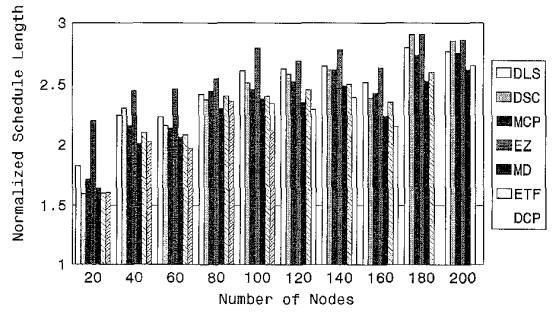


Fig. 12. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for fork-join graph; algorithm ranking: DCP, MD, ETF, MCP, DSC, DLS, EZ.

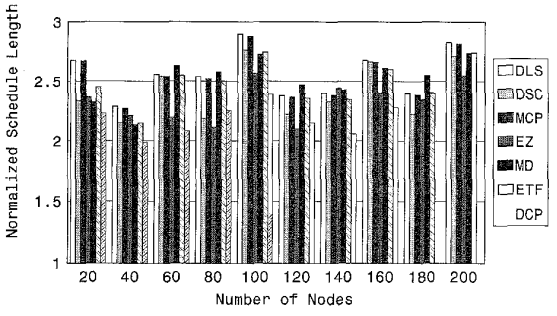


Fig. 13. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for in-tree graphs; algorithm ranking; DCP, EZ, DSC, ETF, MD, MCP, DLS.

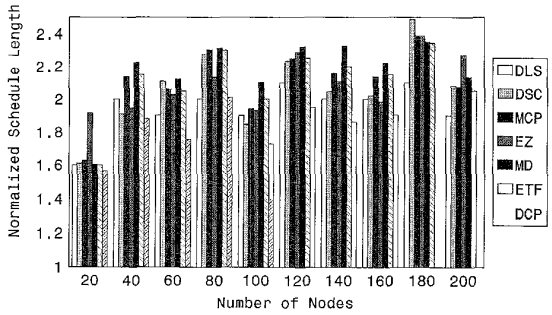


Fig. 14. Average normalized schedule lengths (with respect to lower bounds) at various graph sizes for out-tree graphs; algorithm ranking; DCP, DLS, DSC, (ETF, MCP, EZ), MD.

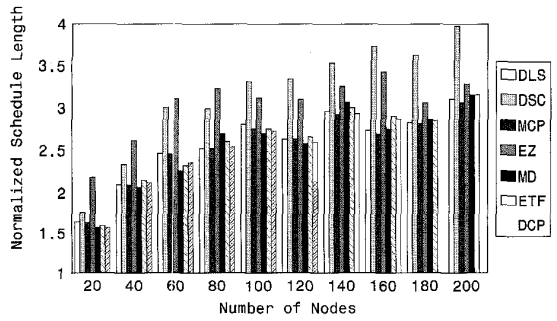


Fig. 15. Average normalized schedule lengths (with respect to lower bounds at various graph sizes for completely random graphs; algorithm ranking: DCP, MD, (MCP, DLS, ETF), EZ, DSC.

- *Mean value analysis graphs.* For the mean value analysis task graphs, all CP-based algorithms, except MD, performed better (see Fig. 11). Similar to the FFT task graphs, all paths are CPs but they intervene largely with each other. MD cannot handle this kind of situation efficiently due to its processor selection criterion. ETF, which gave a similar performance as MCP, performed the best among other non-CP-based algorithms.
- *Fork-join graphs.* For the fork-join task graphs, all CP-based algorithms performed considerably better than the other two algorithms (see Fig. 12). As discussed above, the reason is that there is only one CP in every fork-join task graph. Here, ETF once again performed the best among other non-CP-based algorithms.
- *In-tree graphs.* For the in-tree task graphs, with the exception of the DCP algorithm, the other CP-based algorithms did not perform well (see Fig. 13). This is because although there is only one CP in each in-tree task graph, there are many inward algorithm can schedule different sections of the CPs to different nodes incident on the CP. The DCP processors to avoid delaying their start times by an efficient scheduling of the inward nodes to the CP. The other CP-based algorithms do not handle these cases very well because they try to start every node on the CP as early as possible.
- *Out-tree graphs.* In each out-tree task graph, there is no inward node but many outward nodes emerging from the single CP. Again, the other CP-based algorithms did not perform well because they tend to schedule the only CP to one processor. The results shown in Fig. 14 indicate that the final schedule lengths do not always depend on the CP only but also on the outward nodes from the CP.
- *Completely random graphs.* For completely random task graphs, the DSC and EZ algorithms performed worse in general compared with other algorithms (see Fig. 15). DSC performed considerably worse in quite a number of cases due to the fact that the re-scheduling process can severely block the subsequent scheduling of lower level nodes on the CP.

In general, we can conclude from the above observation that when a task graph has only a few intervening CPs, the CP-based algorithms can perform better. On the other hand, if the task graph contains many CPs, the CP-based algorithms can be "confused" by a particular CP in that the algorithms attempt to start all nodes on that CP as early as possible without noting that the nodes on other intervening CPs are delayed. The DCP algorithm tackles this drawback because it always performs a looking ahead processor selection so that it can avoid being confused by a particular CP.

6.3 A Global Comparison

In order to rank all the algorithms in terms of the scheduled lengths, we made a global comparison in which we observed the number of times each algorithm performed better, worse or the same compared to each of the other six algorithms. This comparison is given in a graphical form shown in Fig. 16. Here, each box compares two algorithms—the algorithm on the left side and the algorithm on the top. Each comparison is based on a total of 966 task graphs which were generated by using the combination of all of the graph structures mentioned above with various number of nodes and CCRs. Each box contains three numbers preceded by ">", "<", and "=" signs which indicate the number of times the algorithm on the left performed better, worse, and the same, respectively, compared to the algorithm shown on the top. For example, the DCP algorithm performed better than the MD algorithm in 699 cases, performed worse in 73 cases, and performed the same in 194 cases. Similarly, the DSC algorithm performed better than the DLS algorithm in 362 cases, performed worse in 427 cases, and performed the same in 177 cases. An additional box for each algorithm compares that algorithm with all other algorithms combined. We can notice that the proposed DCP algorithm outperformed all other algorithms. Furthermore, the numbers given in Fig. 16 indicate that the difference between the performance of DCP and other algorithms was much higher compared to the difference between the performance of the other six algorithms when compared amongst each other. Based on these experiments, we can rank all seven algorithms in the following order: DCP, MCP, DLS, ETF, MD, DSC, and EZ. It should be noted that ETF gave a performance close to DLS, even though the complexity of DLS is higher.

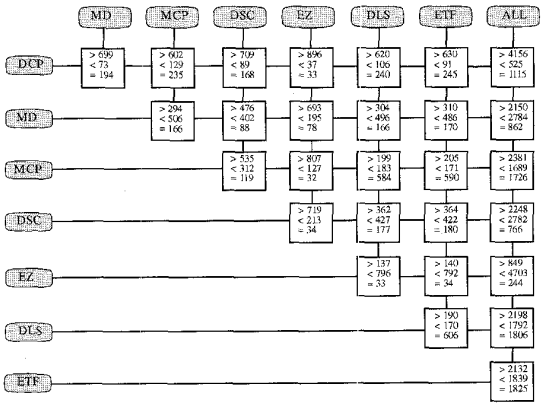


Fig. 16. A global comparison of the seven algorithms in terms of better, worse and equal performance.

6.4 Number of Processors

Another quality of measure for a scheduling algorithm is the number of processors used because each algorithm "spends" a processor in a different way.[2] Fig. 17 shows the average number of processors used by each algorithm for different graph sizes. These averages were taken across all types of task graphs and values of CCR. We observe that DSC used considerably large number of processors compared to the other algorithms while DLS and MCP used approximately the same number of processors. Here, MD outperformed all other algorithms while DCP used slightly more processors than MD. However, this is due to the deficiency of MD because it tries to cluster task on fewer processors. As a results, the schedules generated by MD are not very well load balanced. On the other hand, DCP overcomes this deficiency of MD and produces better schedule lengths by performing some load balancing at the expense of a few more processors.

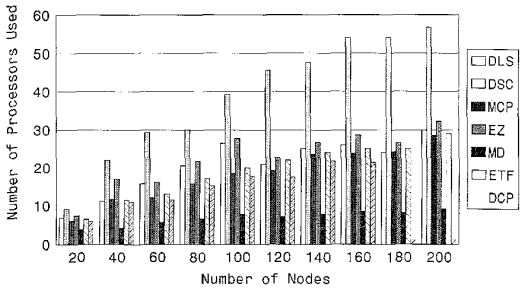


Fig. 17. Average number of processors used by each algorithm; algorithm ranking; MD, DCP, MCP, ETF, DLS, EZ, DSC.

6.5 Comparison of Running Times

Finally, we compare the running times of these algorithms which are given in Fig. 18. From this figure, we can immediately notice that DLS is slower than the other algorithms. It should be noted that the version of DLS used by us was the one that generates the best solution but has a higher complexity. Both DSC and MCP are low complexity algorithms. However, they do not always produce short schedule lengths. The running times of DCP were comparable to MD but more than DSC and MCP. However, the running times of DCP were admissible. Note that the algorithm ranking shown in Fig. 18 is consistent with the given complexities of these algorithms.

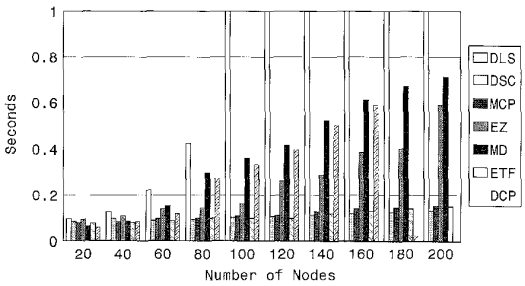


Fig. 18. Average running time for each algorithm; algorithm ranking; DSC, ETF, MCP, EZ, DCP, MD, DLS.

2. Although MCP, DLS, and ETF assume a limited number of processors, they are given a very large number of processors in our experiment so that this is not a limiting factor to their performance.

7 Conclusions

In this paper, we have presented a new scheduling algorithm which outperforms six other algorithms. The difference between the performance of our algorithm and the other algorithms is also much higher than the difference between the performance of other algorithms when compared against each other. The proposed algorithm works better on various types of graph structures. The number of processors used and the running time of the proposed algorithm makes it a viable choice for static compile-time scheduling of macro-data flow graphs and other task graphs onto multiprocessors. The proposed algorithm in its present form assumes a network of fully connected processors but can be generalized to other networks such as hypercube, mesh, etc. In order to accomplish that, the procedure for computing the start times of nodes on the processors will need to be modified and it will need to take into account the hop distances of the processors holding the parent nodes.

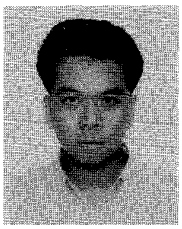
Acknowledgments

We thank Min-You Wu for his help on understanding the MD algorithm. We are also very thankful to the referees, whose constructive comments and suggestions have greatly improved the quality of this paper.
This research was supported by the Hong Kong Research Grants Council under contract number HKUST 179/93E.

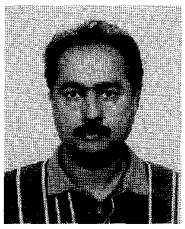
References

[1] T.L. Adam, K. Chandy and J. Dickson, "A Comparison of List Scheduling for Parallel Processing Systems," *Comm. ACM*, vol. 17, no. 12, pp.685-690, Dec. 1974.
[2] V.A.F. Almeida, I.M. Vasconcelos, J.N.C. Arabe and D.A. Menasce, "Using Random Task Graphs to Investigate the Potential Benefits of Heterogeneity in Parallel Systems," *Proc. Supercomputing*, pp. 683-691, 1992.
[3] M.A. Al-Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs with Communication Costs," *IEEE Trans. Software Engineering*, vol. 16, no. 12, pp. 1,390-1,401, 1990.
[4] F.D. Anger, J.J. Hwang, and Y.C. Chow, "Scheduling with Sufficient Loosely Coupled Processors," *J. Parallel and Distributed Computing*, vol. 9, pp. 87-92, 1990.
[5] A.F. Bashir, V. Susarla, and K. Vairavan, "A Statistical Study of the Performance of a Task Scheduling Algorithm," *IEEE Trans. Computers*, vol. 32, no. 12, pp. 774-777, Dec. 1975.
[6] S. Bokhari, "A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in Distributed Processor Systems," *IEEE Trans. Software Engineering*, vol. 7, no. 6, Nov. 1981.
[7] J. Bruno, E.G. Coffman, and R. Sethi, "Scheduling Independent Tasks to Reduce Mean Finishing Time," *Comm. ACM*, vol. 17, no. 7, pp. 382-387, July 1974.
[8] V. Chaudhary and J.K. Aggarwal, "A Generalized Scheme for Mapping Parallel Algorithms," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 3, pp. 328-346, Mar. 1993.
[9] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*. New York: Wiley, 1976.
[10] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, "Parallel Gaussian Elimination on an MIMD Computer," *Parallel Computing*, vol. 6, pp. 275-296, 1988.
[11] E.B. Fernadez and B. Bussell, "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules," *IEEE Trans. Computers*, vol. 22, no. 8, pp. 745-751, Aug. 1973.
[12] M.R. Gary and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[13] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAG's on Multiprocessors," *J. Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276-291, Dec. 1992.
[14] M.J. Gonzalez, 'Deterministic Processor Scheduling," *ACM Computing Surveys*, vol. 9, no. 3, pp. 173-204, Sept. 1977.
[15] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnoy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics*, no. 5, pp. 287-326, 1979.
[16] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *SIAM J. Computing*, vol. 18, no. 2, pp. 244-257, Apr. 1989.
[17] D.S. Hochbaum and D.B. Shmoys, "Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results," *J. ACM*, vol. 34, no. 1, pp. 144-162, Jan. 1987.
[18] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Oper. Research*, vol. 19, no. 6, pp. 841-848, Nov. 1961.
[19] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, vol. 33, no. 11, pp. 1,023-1,029, Nov. 1984.
[20] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocesor Architectures," *Proc. Int'l Conf. Parallel Processing*, pp. 1-8, 1988.
[21] W.H. Kohler and K. Steiglitz, "Characterization and Theoretical Comparison of Branch-and-Bound Algorithms for Permutation Problems," *J. ACM*, vol. 21, no. 1, pp. 140-156, Jan. 1974.
[22] W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on MultiProcessor Systems," *IEEE Trans. Computers*, vol. 24, no. 12, pp. 1,235-1,238, Dec. 1975.
[23] B. Lee, A.R. Hurson, and T.Y. Feng, "A Vertically Layered Allocation Scheme for Data Flow Systems," *J. Parallel and Distributed Computing*, vol. 11, pp. 175-187, 1991.
[24] S.Y. Lee and J.K. Aggarwal, "A Mapping Strategy for Parallel Processing," *IEEE Trans. Computer*, vol. 36, no. 4, pp. 433-442, Apr. 1987.
[25] T.G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*, New York: Prentice Hall, 1992.
[26] R.E. Lord, J.S. Kowalik, and S.P. Kumar, "Solving Linear Algebraic Equations on an MIMD Computer," *J. ACM*, vol. 30, no. 1, pp. 103-117, Jan. 1983.
[27] C. McCreary and H. Gill, "Automatic Determination of Grain Size for Efficient Parallel Processing," *Comm. ACM*, vol. 32, pp. 1,073-1,078, Sept. 1989.
[28] J.M. Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, New York: Plenum, 1988.
[29] C. Papadimitriou and M. Yannakakis, "Toward an Architecture Independent Analysis of Parallel Algorithms," *SIAM J. Computing*, vol. 19, pp. 322-328, 1990.
[30] C.V. Ramamoorthy, K.M. Chandy, and M.J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. Computers*, vol. 21, no. 2, pp. 137-146, Feb. 1972.
[31] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Programs onto Arbitrary Target Machines," *J. Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138-153, June 1990.
[32] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Cambridge, Mass: MIT Press, 1989.
[33] R. Sethi, "Scheduling Graphs on Two Processors," *SIAM J. Computing*, vol. 5, no.1, pp. 73-82, Mar. 1976.
[34] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Scheduling," *J. Parallel and Distributed Computing*, no. 10, pp. 222-232, 1990.
[35] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 2, pp. 75-187, Feb. 1993.
[36] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 9, Sept. 1994.
[37] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, July 1990.



Yu-Kwong Kwok (S'94) is a PhD candidate in the Department of Computer Science, at the Hong Kong University of Science and Technology. He received his BSc degree in computer engineering from the University of Hong Kong in 1991, and his MPhil degree in computer science from the Hong Kong University of Science and Technology in 1994. His research interests include algorithms, parallel and distributed computing, parallel programming languages, and compilers. He is a member of the IEEE Computer Society, and the ACM.



Ishfaq Ahmad received his BSc degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan in 1985, the MS degree in computer engineering, and his PhD in computer science, from Syracuse University, in 1987 and 1992, respectively. Currently, Dr. Ahmad is a faculty member in the Department of Computer Science at the Hong Kong University of Science and Technology. His research interests include various aspects of parallel and distributed computing, high-performance computer architectures and their assessment, and performance evaluation. He received the Best Student Paper Awards at Supercomputing '90 and Supercomputing '91. He has been a guest editor for two special issues of *Concurrency: Practice and Experience*, and has served on the program committees of various international conferences. Dr. Ahmad is a member of the IEEE Computer Society.