On Exploiting Task Duplication in Parallel Program Scheduling

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

Abstract—One of the main obstacles in obtaining high performance from message-passing multicomputer systems is the inevitable communication overhead which is incurred when tasks executing on different processors exchange data. Given a task graph, duplication-based scheduling can mitigate this overhead by allocating some of the tasks redundantly on more than one processor. In this paper, we focus on the problem of using duplication in static scheduling of task graphs on parallel and distributed systems. We discuss five previously proposed algorithms and examine their merits and demerits. We describe some of the essential principles for exploiting duplication in a more useful manner and, based on these principles, propose an algorithm which outperforms the previous algorithms. The proposed algorithm generates optimal solutions for a number of task graphs. The algorithm assumes an unbounded number of processors. For scheduling on a bounded number of processors, we propose a second algorithm which controls the degree of duplication according to the number of available processors. The proposed algorithms are analytically and experimentally evaluated and are also compared with the previous algorithms.

Index Terms—Algorithms, distributed systems, multiprocessors, duplication-based scheduling, parallel scheduling, task graphs.

1 Introduction

DESPITE great advances in multicomputer architecture design, interprocessor communication remains a notoriously unavoidable overhead in the execution of parallel programs. This overhead is incurred when tasks of the parallel program assigned to different processors exchange data. Since the communication cost between tasks assigned to the same processor is considered to be negligible, task duplication is one way of reducing the interprocessor communication overhead. Using this approach, some of the more critical tasks of a parallel program are duplicated on more than one processor. This can potentially reduce the start times of waiting tasks and eventually improve the overall completion time of the entire program. Duplication-based scheduling can be useful for systems, such as networks of workstations, that have high communication latencies and low bandwidths. With task duplication, considerable improvements in speedups have been reported [14].

For duplication-based scheduling, the structure of a parallel program and the timings of individual tasks and communication costs must be available (see [18], [23] for techniques to generate such information). Therefore, scheduling can be performed statically at compile-time. A parallel program represented by a directed acyclic graph (DAG) contains v nodes $\{n_1, n_2, \ldots, n_v\}$ and e directed edges, each of which is denoted by (n_i, n_j). A node in the parallel program graph represents a task which, in turn, is a set of instructions that must be executed sequentially in the same processor (we assume no preemption). Associated with each node is the *computation cost* of a node n_i, denoted by $w(n_i)$. The directed edges in the parallel program graph correspond to the communication messages as well as precedence constraints among the tasks. Associated with each edge is the *communication cost* of the edge, denoted by $c(n_i, n_j)$. The source node of an edge is called the *parent* node, while the destination node is called the *child* node. A node without a parent is called an *entry* node, while a node without a child is called an *exit* node. 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. A node cannot start execution before it gathers all of the messages from its parent nodes. If node n_i is scheduled to processor P, $ST(n_i, P)$ and $FT(n_i, P)$ denote the start time and finish time of n_i on processor P, respectively. It should be noted that $FT(n_i, P) = ST(n_i, P) + w(n_i)$. After all nodes have been scheduled, the schedule length is defined as $\max_i\{FT(n_i, P)\}$ across all processors. The objective of a scheduling algorithm is minimize the schedule length such that the precedence constraints are preserved.

Scheduling of task graphs onto multiprocessors is known to be an NP-complete problem in most cases [6], [11], [14], [17], leading to solutions based on heuristics [9], [10], [16], [21]. The complexity and quality of a heuristic largely depend on the task graph structure and the target machine model. The conventional heuristic used in designing scheduling algorithms is called *list scheduling*, which is a two-step approach. In the first step, priorities are assigned to nodes and 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,

- *I. Ahmad is with the Department of Computer Science, The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: iahmad@cs.ust.hk.*
- *Y.-K. Kwok is with the Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: ykwok@eee.hku.hk.*

Manuscript received 16 Feb. 1996; revised 1 Apr. 1997.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 100122.

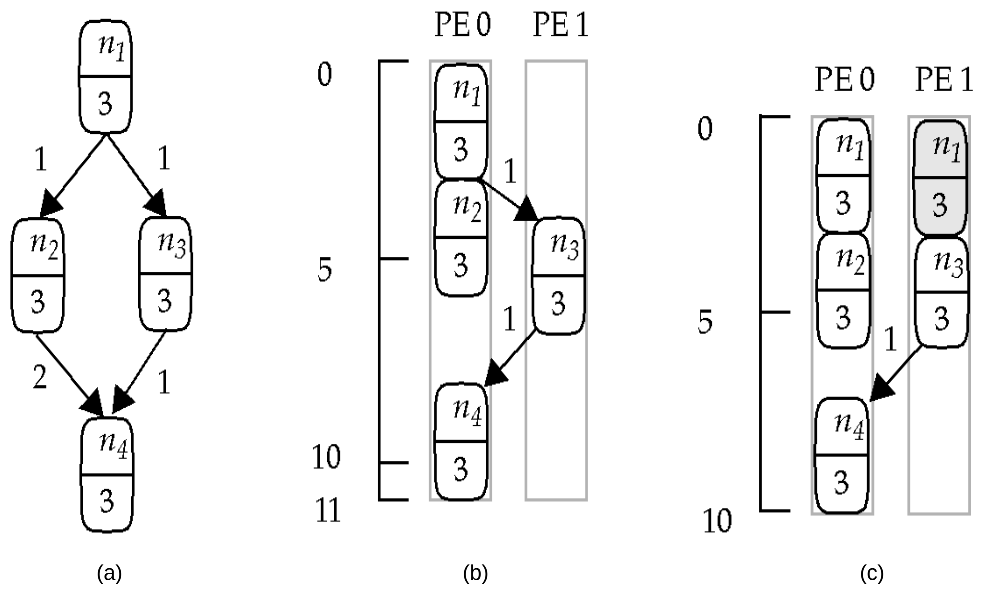


Fig. 1. (a) A simple task graph, (b) a schedule without duplication, (c) a schedule with duplication.

is selected to accommodate this node. Numerous methods for assigning priorities to nodes and selecting the most suitable processor have been proposed [7], [8], [13], [19], [22].

Even with an efficient scheduling algorithm, it may happen that some processors are idle during different time slots because some nodes must wait for data from nodes assigned to other processors. If these idle time slots are utilized effectively by identifying and redundantly allocating the critical nodes, the completion time of the parallel program can be further reduced. Consider, for instance, a simple task graph shown in Fig. 1a. An optimal schedule without duplication is shown in Fig. 1b (PE denotes a processing element). As can be seen, PE 1 is idle from time 0 to time 4 since node n_3 is waiting for the output data from n_1. If n_1 is duplicated to this idle time period of PE 1, the schedule length can be reduced to the minimum, as shown in Fig. 1c. There are two issues to be addressed for designing an effective task duplication technique:

- *Which node(s) to duplicate?* This concerns the selection of the ancestor nodes for duplication so that the finish time of a descendent node can be minimized.
- *Where to duplicate the node(s)?* This concerns locating a proper time slot on a processor to duplicate the ancestor nodes.

One might argue that task duplication requires more memory. However, even if all ancestors (starting from an entry node) of a node are duplicated, the sum of the computation costs of the duplicated nodes does not exceed the critical path[1] length, which is the maximum memory requirement per processor without duplication. Thus, duplication has the potential to considerably reduce the schedule length by efficiently utilizing the processors, provided an efficient scheduling algorithm is available. Using duplication, however, makes the scheduling problem more complex since the scheduling algorithm should not only observe the precedence constraints but also select important ancestor nodes for duplication and identify idle time slots to accommodate them.

This paper is organized as follows. In Section 2, we describe the related work in duplication-based scheduling through a discussion of the characteristics of five previously proposed algorithms. This discussion motivates the need for a more effective algorithm. In Section 3, we present the design principles of our approach, followed by a description of our proposed algorithms and some of their properties. In Section 4, we present some illustrative examples to demonstrate the operation of all the algorithms. In Section 5, we include the performance results and comparisons with other algorithms. We provide concluding remarks in the last section, and include proofs of theorems in the Appendix.

2 Related Work

Using duplication in static scheduling is a relatively unexplored research topic and only a few such algorithms have been suggested in the literature. The main difference in the reported algorithms lies in their strategies to select nodes for duplication. To reduce the start times of nodes, some algorithms duplicate only the parent nodes, while some algorithms attempt to duplicate ancestor nodes from higher levels as well. In the following, we discuss five duplication-based scheduling algorithms (Table 1 includes some of the characteristics of these algorithms).

2.1 The DSH Algorithm

The DSH (Duplication Scheduling Heuristic) algorithm [12] uses the *static level* (defined as the largest sum of computation costs along a path from the node to an exit node) as the priority for each node. The algorithm considers each node in descending order of their priorities. In examining the suitability of a processor for a node, the algorithm first

1. A critical path of a task is a set of nodes and edges, forming a path from an entry node to an exit node, of which the sum of computation cost and communication cost is the maximum; there can be more than one critical path in a task graph.

TABLE 1
SOME DUPLICATION-BASED SCHEDULING ALGORITHMS AND THEIR CHARACTERISTICS

Algorithm	Proposed by [year]	Ancestors Duplicated	Complexity
DSH	Kruatrachue & Lewis [1987]	All possible ancestors	$O(v^4)$
PY	Papadimitriou & Yannakakis [1990]	All possible ancestors	$O(v^2(e + v\log v))$
LWB	Colin & Chretienne [1990]	Only ancestors on the same path	$O(v^2)$
BTDH	Chung & Ranka [1992]	All possible ancestors	$O(v^4)$
LCTD	Chen *et al.* [1993]	All possible ancestors	$O(v^3\log v)$

determines the start time of that node on the processor *without* duplication of any ancestor, and then considers the duplication time slot (the idle time period from the finish time of the last scheduled node on the processor and the start time of the node currently under consideration). If a suitable processor is found, the algorithm attempts to duplicate the parents of the node into the duplication time slot until either the slot is used up or the start time of the node does not improve further. This process is repeated for other processors and the node is scheduled to the processor that gives the smallest start time. The DSH algorithm calculates the priorities of nodes based on static levels which may not accurately capture the relative importance of nodes because dynamically changing communication costs (during the scheduling steps) among nodes are not taken into account. Furthermore, duplication may not always be very effective since the algorithm considers only one idle time slot on a processor.

2.2 The PY Algorithm

The PY algorithm (named after Papadimitriou and Yannakakis) [17] uses an attribute called the *e-value* to approximate the absolute achievable lower bound of the start time of a node. This attribute is computed recursively beginning from the entry nodes to the exit nodes. After computing the *e-values*, the algorithm inserts each node into a cluster in which a group of parents are duplicated such that the data arrival times from these ancestors are larger than the *e-value* of the node. It has been shown that the schedule length generated by the algorithm is within a factor of two from the optimal. The algorithm clusters nodes in a subgraph for duplication by using a node inclusion inequality which checks the message arrival times against the lower bound values of the candidate node under consideration. This can potentially leave out the nodes which are more important for reducing the start time of the given node, and this may lead to a poor schedule.

2.3 The LWB Algorithm

We call the algorithm proposed in [5] the LWB (Lower Bound) algorithm based on its main procedure. The algorithm first determines the lower bound start time (denoted by *lwb*) for each node and then identifies a set of critical edges in the DAG. A critical edge is one in which a parent's message available time for the child is greater than the lower bound start time of the child. Thus, the parent and child have to be scheduled to the same processor in order to reduce the start time of the child. Based on this idea, the LWB algorithm schedules every path of critical edges to a distinct processor. Since these paths may share ancestors, duplication is employed. It should be noted that the *lwb* value of a node is different from the *e*-value used in the PY algorithm in that the *lwb* value is computed by considering a single path from an entry node, while the *e*-value is computed by taking the whole subgraph reaching the node into account. The algorithm considers only those ancestors which are on a single path. When a node has more than one heavily communicated parent, this technique does not minimize the start time of the node (which can be done by duplicating more than one parents on a processor). Nevertheless, as is shown in [5], the LWB algorithm can generate optimal schedules for task graphs in which node weights are strictly larger than any edge weight.

2.4 The BTDH Algorithm

The BTDH (Bottom-Up Top-Down Duplication Heuristic) algorithm [4] is essentially an extension of the DSH algorithm described above. The major improvement brought by the BTDH algorithm over the DSH algorithm is that the former keeps on duplicating ancestors of a node even when the duplication time slot is filled up and the start time of the node under consideration temporarily increases. This strategy is based on the intuition that the start time may eventually be reduced by duplicating all the necessary ancestors. As the BTDH algorithm also uses *static level* for priority assignment, it may not always accurately capture the relative importance of nodes.

2.5 The LCTD Algorithm

The LCTD (Linear Clustering with Task Duplication) algorithm [20] first iteratively clusters nodes into larger nodes. At each iteration, nodes on the longest path are clustered and removed from the task graph. This operation is repeated until all nodes in the graph are removed. After performing the clustering step, the LCTD algorithm identifies those edges among clusters that determine the overall completion time. The algorithm then attempts to duplicate the parents corresponding to these edges to reduce the start times of some nodes in the clusters. Linear clustering may not always accurately identify the nodes that should be scheduled to the same processor. In addition, in the context of duplication based scheduling, linear clustering prematurely constrains the number of processors used. This constraint can be detrimental because the start times of some critical nodes may possibly be significantly reduced by using a new processor in which its ancestors are duplicated.

3 The Proposed Approach

In this section, we discuss some of the basic principles used in the proposed approach, followed by a description of our algorithms. The first algorithm assumes unbounded number of processors while the second algorithm takes the number of processors as an input parameter.

The assumptions in our approach are the same as those in the other algorithms mentioned earlier: We assume that the processor network is fully connected and each processor has a dedicated communication hardware so that communication and computation can take place simultaneously.

3.1 Assigning Priorities to Nodes

An accurate determination of important nodes for duplication is the key to obtaining a short schedule. If relatively less important nodes are scheduled or duplicated, the early time slots in the processors are occupied. Consequently, the more important nodes are not scheduled to start earlier. The most important nodes are the nodes on the critical path (CP). This is because a CP is the longest path of the task graph and, therefore, the finish times of the CP nodes (CPNs) determine the final schedule length. Thus, CPNs should be examined for scheduling or duplication as early as possible. However, not all CPNs can be examined without considering their parent nodes because of precedence constraints. In order to formulate a scheduling order in which all the CPNs can be scheduled as early as possible while preserving precedence constraints, we classify the nodes of a task graph into three categories using the following definition.

Definition 1. *In a connected graph, an In-Branch Node* (*IBN*) *is a node which is not a CPN and from which there is a path reaching a Critical Path Node* (*CPN*). *An Out-Branch Node* (*OBN*) *is a node which is neither a CPN nor an IBN*.

The relative importance of these nodes is in the following order: CPNs, IBNs, and OBNs. The IBNs are also important because timely scheduling of these nodes can help reducing the start times of the CPNs. The OBNs are relatively less important because they usually do not affect the schedule length. Based on this classification, we construct a list of nodes in decreasing importance so that the CPNs are examined for scheduling before the other nodes without violating the precedence constraints. This list of nodes is called the *CPN-Dominant list* and is built in the following manner:

Construction of the CPN-Dominant list:

```
1)    Initially, the list is empty. Make the entry CPN be the first node in the list.
      Set Position to 2. Let $n_x$ be the next CPN.
   Repeat
2)    If $n_x$ has all its parent nodes in the list then
3)        Put $n_x$ at Position in the list and increment Position.
4)    else
5)        Let $n_y$ be the parent node of $n_x$ which is not in the sequence and has the
          largest b-level.[2] Ties are broken by choosing the parent with a smaller t-level.
          If $n_y$ has all its parent nodes in the sequence, put $n_y$ at Position in the
          sequence and increment Position. Otherwise, recursively include all the ancestor
          nodes of $n_y$ in the sequence so that the nodes with a larger value of b-level
          are considered first.
6)        Repeat the above step until all the parent nodes of $n_x$ are in the list. Then,
          put $n_x$ at Position in the list.
7)    endif
8)    Make $n_x$ to be the next CPN.
   Until all the CPNs are in the list.
9)    Append all the OBNs to the sequence in a decreasing order of b-level.
```

The CPN-Dominant sequence does not violate the precedence constraints since all of the IBNs reaching a CPN are always before that CPN in the sequence. In addition, the OBNs are appended to the sequence in a topological order so that a parent OBN is always before a child OBN.

3.2 The Task Duplication Technique

Locating a proper time slot to accommodate a duplicated node is also very important since duplicating a node in an improper time slot may not reduce the start time of a node. Some previously proposed algorithms consider only the last idle time slot on a processor but ignore other idle time slots that may give more reduction in the schedule length.

The start time of a node (call it a *candidate* node) is determined by the data arrival time from its parent nodes. The parent node from which the data arrives at the latest time is called a *Very Important Parent* (VIP) of the candidate node. Rule I below can be used to determine the VIP of a node.

Rule I. *The Very Important Parent* (*VIP*) *of a node n_y is a parent node n_x such that the value of $FT(n_x, MINP(n_x)) + c(n_x, n_y)$ is the largest among all the n_y's parent nodes. This value is called the data arrival time* (*DAT*) *of n_y.*

To accommodate this VIP, a proper time slot on a processor must be found. Our approach is to scan through the whole time span of the processor to find the *earliest* time slot that is large enough to accommodate the selected parent node provided the precedence constraints are not violated. Rule II below governs the selection of a suitable idle time slot.

Rule II. *A node n_y can be scheduled to a processor P, on which a set of k nodes $\{n_1^P, n_2^P, \ldots, n_k^P\}$ have been scheduled, if there exists some value of i such that:*

$$ST(n_{i+1}^P, P) - \mathrm{MAX}\{FT(n_i^P, P), DAT(n_y, P)\} \geq w(n_y)$$

where i = 0, 1, …, k; $ST(n_{k+1}^P, P) = \infty$, and $FT(n_0^P, P) = 0$.

2. The *b-level* of a node is the length (sum of the computation and communication costs) of the longest path from this node to an exit node. The *t-level* of a node is the length of the longest path from an entry node to this node (excluding the cost of this node).

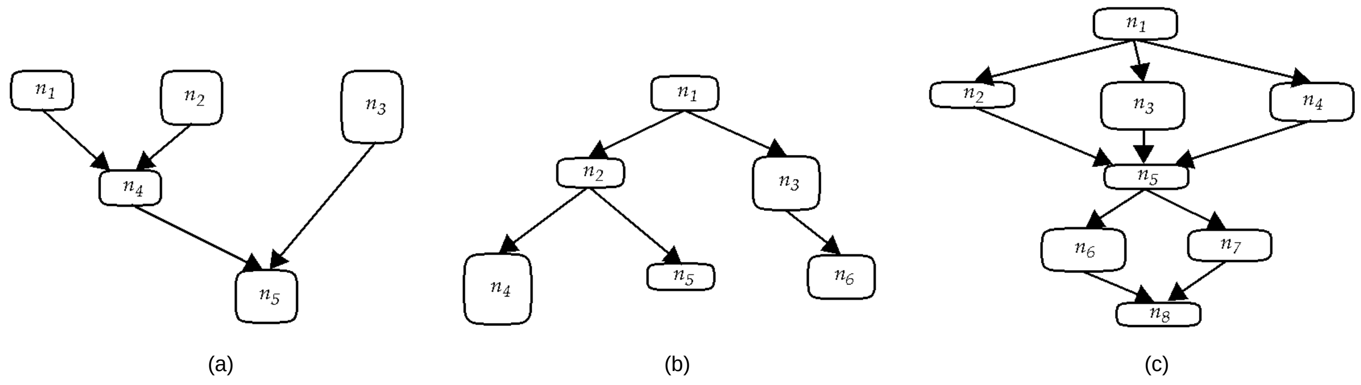


Fig. 2. (a) An in-tree task graph, (b) and out-tree task graph, (c) a fork-join task graph.

The earliest start time of n_y on P is then given by:

$$\mathrm{MAX}\left\{FT\left(n_j^P, P\right), DAT\left(n_y, P\right)\right\}$$

with j being the smallest value of i satisfying the inequality.

Minimization of the start time of a candidate node, in turn, requires the minimization of the start time of its duplicated VIP. This requires the duplication process to be applied *recursively* to the VIP to minimize its start time. Below is an outline of the procedure for performing the recursive duplication process.

Minimize_start_time(*node, P*):

1) Determine the earliest start time (EST) of *node* on processor *P* using Rule II.
2) If EST is undefined, then quit because *node* cannot be scheduled on *P*.
3) Find out the VIP of *node* using Rule I.
4) Minimize the start time of this VIP by recursively calling *Minimize_start_time*(VIP, *P*).
5) Compute the new EST of *node*. If the new EST does not improve, then undo all the duplication just performed in Step 4, schedule *node* to *P*, and quit; otherwise, find out the new VIP of *node* and repeat from Step 4.

The procedure *Minimize_start_time* minimizes the start time of a candidate node by recursively duplicating the ancestor nodes beginning from its VIP. After exploring the ancestor nodes reachable from this VIP, the procedure attempts to further reduce the start time of the candidate node by considering if there is another parent node of the candidate node (the new VIP) that is constraining the start time of the candidate node. The duplication process is applied if such a new VIP exists. Indeed, the duplication process terminates when there is no more VIP or the VIP is already on the same processor.

3.3 The Proposed Algorithm

Using the procedures discussed above, the proposed algorithm, called the Critical Path Fast Duplication (CPFD) algorithm, is formalized below.

The Critical Path Fast Duplication Algorithm:

1) Determine a CP. Ties are broken by selecting the one with a larger sum of computation costs. Construct the CPN-Dominant sequence.

Repeat

2) Let *candidate* be the first unscheduled CPN in the CPN-Dominant sequence.
3) Let *P_SET* be a set of processors, including all the processors holding *candidate*'s parent nodes, and an empty processor.
4) For each processor *P* in *P_SET*, call *Minimize_start_time* (*candidate, P*) and record the EST of *candidate* on *P*.
5) Schedule *candidate* to the processor *P*' that gives the smallest value of EST.

Until all the CPNs in CPN-Dominant sequence are scheduled.

6) Repeat the process from Step 2 to Step 5 for each OBN in the CPN-Dominant sequence.

The scheduling of the IBNs reaching a CPN is implicitly handled by the duplication process in the procedure *Minimize_start_time*. If an IBN is found to be unscheduled during the minimization of the start time of a CPN, the start time of the IBN can always be minimized by scheduling it to an empty processor.

The time complexity of the CPFD algorithm is determined as follows. The dominant part of the algorithm is the loop from Step 2 to Step 5. This loop executes $O(v)$ times as there are $O(v)$ CPNs in a task graph. In each iteration of the loop, there are $O(v)$ calls to the procedure *Minimize_start_time* because there are $O(v)$ processors in *P_SET*. The time complexity of *Minimize_start_time* is $O(e)$ since the procedure traverses $O(e)$ edges in the task graph. Thus, the overall time complexity is $O\left(ev^2\right)$.

3.4 Properties of the CPFD Algorithm

In this section, we examine the theoretical performance of the CPFD algorithm for three types of graph structures: the out-tree task graphs, the in-tree task graphs, and the fork-join task graphs (see Fig. 2). First, we briefly characterize the in-tree task graph, the out-tree task graph, and the fork-join task graph.

- An out-tree task graph is a connected graph in which every node has only one parent node. This task graph represents a number of divide-and-conquer algorithms as the flow of control in these algorithms is usually in a top-down fashion.
- An in-tree task graph is a connected graph in which every node has only one child node. An in-tree task

graph can represent some divide-and-conquer algorithms in which the flow of control is in a bottom-up fashion.

- A fork-join task graph is a hybrid of an in-tree task graph and an out-tree task graph. It has a root node (with depth 0) which spawns a number of children nodes. The output edges of the children nodes connect to either an intermediate node that spawns another set of children nodes or an exit node.

In the following, the notion of *optimal solution* implies the best solution that can be realized *with* duplication. Obviously, the best schedule length with duplication must be shorter than or equal to the best schedule length without duplication.

THEOREM 1. *The CPFD algorithm generates optimal schedules for out-tree task graphs.*

PROOF. Refer to the Appendix. □

Theorem 1 reveals that duplication may also be useful for task graphs having nodes with more children nodes than parent nodes, even though the task graphs are not out-tree graphs. In other words, given a task graph with more OBNs than IBNs, the algorithm can accurately identify important tasks for duplication.

THEOREM 2. *The CPFD algorithm generates optimal schedules for any unit-height in-tree task graphs.*

PROOF. Refer to the Appendix. □

The result in Theorem 2 may be achieved with or without task duplication in that the optimal schedule length of the in-tree shown in Fig. 17a can be obtained (see the Appendix) even if we do not apply duplication to the entry nodes. Unfortunately, the result of Theorem 2 cannot be generalized to in-tree task graphs with arbitrary heights. However, we expect that the CPFD algorithm generates near optimal schedules for in-tree task graphs because it recursively traces upward in the task graph and selects the most important parent to duplicate.

THEOREM 3. *The CPFD algorithm generates optimal schedules for any fork-join task graphs.*

PROOF. Refer to the Appendix. □

It should be noted that achieving the results in Theorem 3 requires duplicating the root node n_0 (see the Appendix) on every processors.

If a task graph has relatively small communication costs (such that all communication costs are strictly less than any computation cost in the task graph), the CPFD algorithm can generate optimal schedule regardless of the graph structure.

3.5 Algorithm for Bounded Number of Processors

In this section, we present our second algorithm, called the *Economical CPFD* (CPFD) algorithm, that considers the availability of a bounded number of processors. Before describing the algorithm, we make some observations on the CPFD algorithm. During the scheduling process, it can be noted that some OBNs can be *packed* into a processor already in use instead of using a new processor. Indeed, the start times of some OBNs can be delayed without affecting the schedule length. Thus, we may modify the procedure for scheduling an OBN as follows:

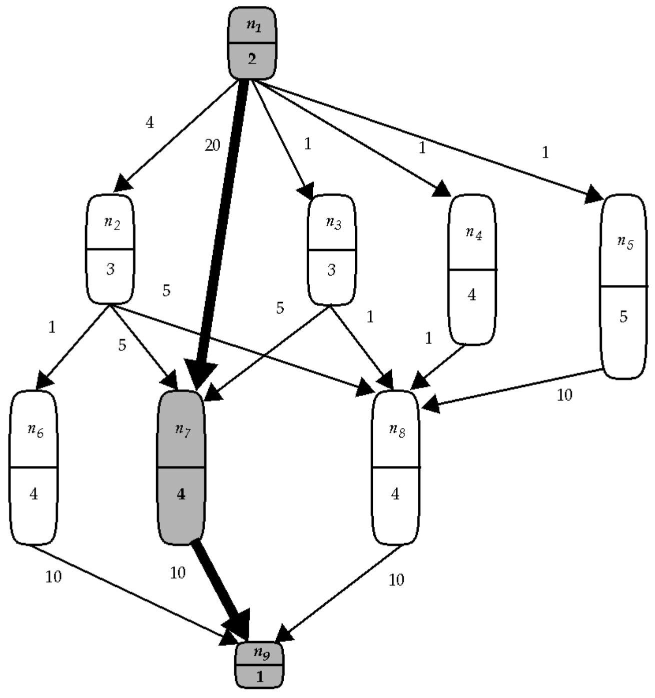
1) Without using any task duplication, schedule the OBN to a processor which is already in use. The OBN is scheduled to start at the earliest start time provided the schedule length does not increase.
2) If none of the processors already used can do the above, schedule the OBN to an empty new processor with duplication so that it can start execution as early as possible. If there is no new processor available, schedule the OBN to a processor already used such that the increase in schedule length is the minimum.

If an OBN is forced to schedule to a processor already used, the start times of its descendants may be delayed and the schedule length will in turn be increased. Thus, we should also check whether the scheduling of the OBN will cause such an increase in schedule length by future scheduling of its descendants. To simplify the procedure we can check only the most important descendant (the one with the largest sum of communication cost and computation cost). The ECPFD algorithm is described below.

The ECPFD Algorithm:

```
1) Determine a CP. Ties are broken by selecting the one
   with a larger sum of computation costs. Construct the
   CPN-Dominant sequence.
Repeat
2) Let candidate be the first unscheduled CPN in the
   CPN-Dominant sequence.
3) Let P_SET be a set of processors including all the
   processors holding candidate's parent nodes and an
   empty processor, if any.
4) For each processor P in P_SET, call Minimize_start_time
   (candidate, P) and record the EST of candidate on P.
5) Schedule candidate to the processor P′ that gives the
   smallest value of EST.
Until all the CPNs in CPN-Dominant sequence are
   scheduled.
Repeat
6) Let candidate be the first unscheduled OPN in the
   CPN-Dominant sequence.
7) Let P_SET be the set of processors already in use.
8) Determine the critical_child of candidate, which is the
   child node that has the largest communication.
9) For each processor P in P_SET:
   a) Call Minimize_start_time(candidate, P)
   b) Call Minimize_start_time(critical_child, P)
   c) Record the sum of the ESTs of candidate and criti-
      cal_child on P.
10) Schedule candidate to the processor P′ that gives the
    smallest sum of ESTs. If no such P′ exists and there is an
    empty processor Q, schedule candidate to Q with dupli-
    cation by calling Minimize_start_time(candidate, Q); oth-
    erwise, schedule candidate to a processor in P_SET such
    that the increase in schedule length is the minimum.
Until all the OBNs in the CPN-Dominant sequence are
   scheduled.
```

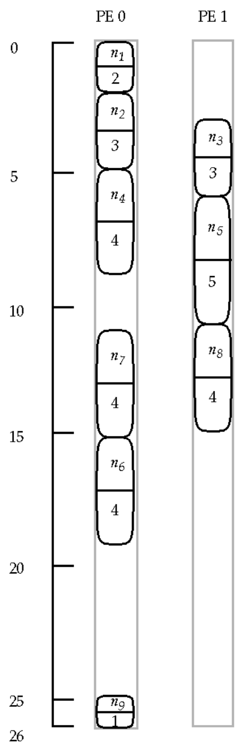
The ECPFD algorithm shares with the CPFD algorithm the basic duplication technique for scheduling CPNs. However, there are also some important differences between



(a)

node	SL	CPN-DS
*n_1	12	1
n_2	8	2
n_3	8	3
n_4	9	6
n_5	10	5
n_6	5	8
*n_7	5	4
n_8	5	7
*n_9	1	9

(b)



(c)

Fig. 3. (a) A simple task graph, (b) static levels of the nodes, (c) a schedule without using task duplication (schedule length = 26).

the two algorithms: The first difference is that since the ECPFD algorithm works with a limited number of processors, some of the CPNs may not be scheduled to start at the earliest possible times if no empty processor is available. The second difference is the way the ECPFD algorithm handles OBNs. Initially, the ECPFD algorithm does not apply any duplication to schedule the OBNs. If none of processors already in use can accommodate an OBN without increasing the intermediate schedule length, the ECPFD algorithm resorts to duplication of that OBN using a new processor. If no new processor is available, the algorithm chooses the processor that can accommodate the OBN with the smallest increase in schedule length. To avoid scheduling an OBN to a processor that cannot accommodate a heavily communicated child node of the OBN, the ECPFD algorithm uses the look-ahead scheduling strategy to select the most suitable processor. As the dominant part of the algorithm is still the loop of scheduling the CPNs, the time complexity of the ECPFD algorithm is $O(pev)$, where p is the number of processors.

4 Scheduling Examples

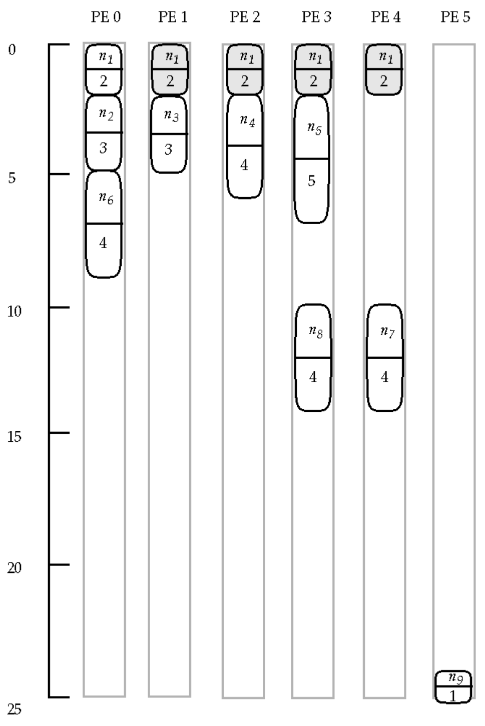
In this section, we first use an example task graph (see Fig. 3a) to illustrate the effectiveness of the proposed algorithms. For comparison, the schedules generated by the other five algorithms are also shown.

For the graph shown in Fig. 3a, the static levels of nodes and the CPN-Dominant Sequence (denoted by CPN-DS) are shown in Fig. 3b. The CPNs in the task graphs are marked by an asterisk. A schedule without task duplication is shown in Fig. 3c. Communication edges are not shown in the schedule for clarity. As can be seen, the node n_3 needs to wait for the data from node n_1, resulting in an idle time period of three units in processor PE 1. Similarly, the node n_7 needs to wait for the data from n_3. Thus, if n_3 can start earlier on PE 1 by duplicating n_1, then n_7 can also start earlier and the schedule length can be reduced.

The schedule generated by the LWB algorithm is shown in Fig. 4, which also includes a scheduling trace. The schedule length is one time-unit shorter compared to the schedule without duplication. However, the number of processors used increases from 2 to 5. Obviously, the duplication employed by the LWB algorithm for this task graph is not effective. This is because the algorithm attempts to duplicate only ancestor nodes on the same path despite that the start time of a candidate node can be further reduced by duplicating the ancestor nodes on the other paths. For instance, the start time of the node n_7 can be considerably reduced if the nodes n_2 and n_3 are also duplicated to PE 4.

The schedule generated by the LCTD algorithm and the scheduling trace are shown in Fig. 5. The schedule length is shorter than that of the LWB algorithm and the utilization of processors is also much better. This is because the LCTD algorithm considers every ancestor nodes reaching a node for duplication.

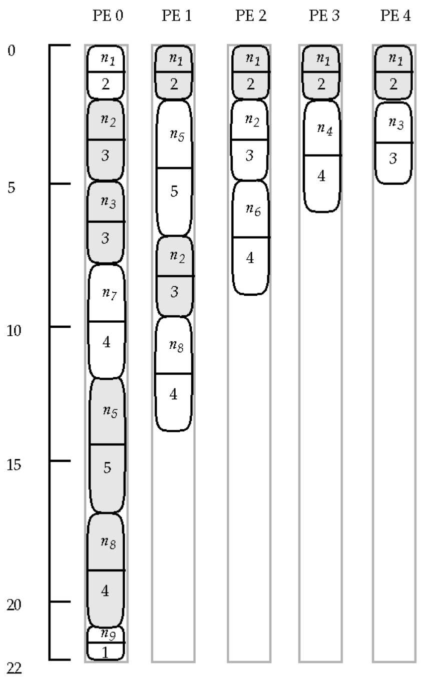
The DSH and BTDH algorithms generate the same schedule, which is shown in Fig. 6. Although the schedule length is the same as that of the LCTD algorithm, the scheduling order of most nodes is different. This is because the LCTD algorithm assigns all the nodes on a critical path to the same processor at once, while the DSH and BTDH



node	*lwb*	Parent of critical edge
n_1	0	NIL
n_2	2	n_1
n_3	2	n_1
n_4	2	n_1
n_5	2	n_1
n_6	5	n_2
n_7	10	n_1
n_8	10	n_4
n_9	24	NIL

(a) (b)

Fig. 4. (a) The schedule generated by the LWB algorithm (schedule length = 25), (b) the lower bound values computed by the LWB algorithm.



Step	Node	Old ST	New ST	Nodes Dup.
1	n_1	0	0	NIL
2	n_7	14	8	n_2, n_3
3	n_9	28	21	n_5, n_8
4	n_5	3	2	n_1
5	n_8	14	10	n_2
6	n_2	6	2	n_1
7	n_6	9	5	NIL
8	n_4	3	2	n_1
9	n_3	3	2	n_1

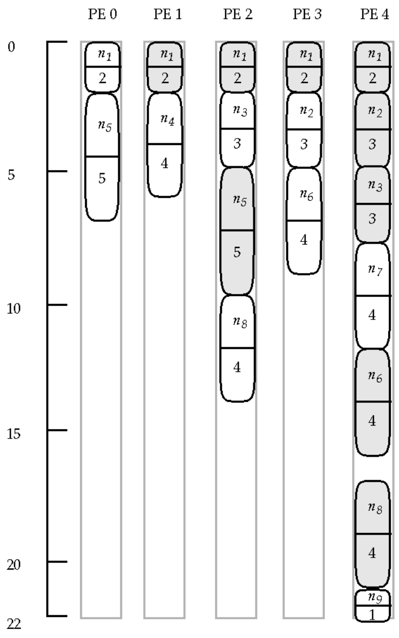
(a) (b)

Fig. 5. (a) The schedule generated by the LCTD algorithm (schedule length = 22), (b) a scheduling trace of the LCTD algorithm.

algorithms examine the nodes for scheduling in a descending order of static levels.

The schedule generated by the PY algorithm is shown in Fig. 7. The schedule length is much longer than that of the previous three algorithms. It should be noted that the schedule length is even longer than the schedule without duplication shown earlier.

The schedule generated by the CPFD algorithm is shown in Fig. 8a. The schedule length is 20, which is optimal. The scheduling steps are also shown in Fig. 8b. Note that this task graph does not contain any OBN. The order of scheduling depicted in the table follows the order in the CPN-Dominant sequence. The nodes duplicated at each step are also shown in the last column of the table. At the first step, the first CPN,

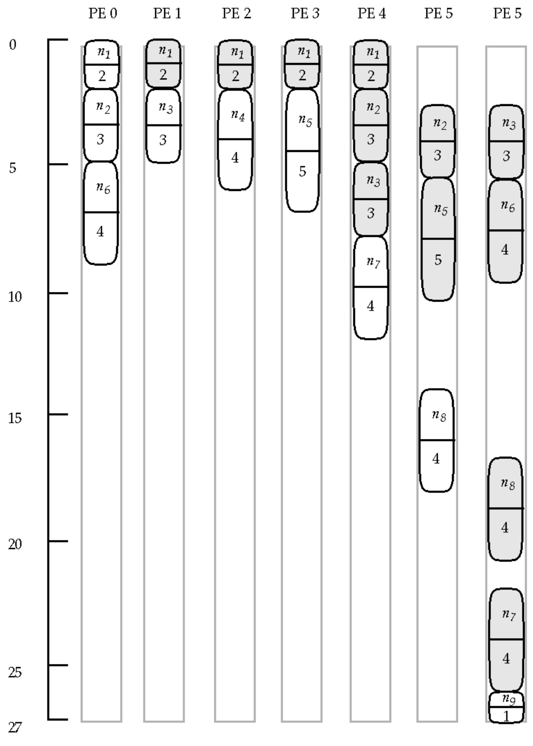

PE 0
PE 1
PE 2
PE 3
PE 4
0
5
10
15
20
22
n_1 2 n_5 5
n_1 2 n_4 4
n_1 2 n_3 3 n_5 5 n_8 4
n_1 2 n_2 3 n_6 4
n_1 2 n_2 3 n_3 3 n_7 4 n_6 4 n_8 4 n_9 1

(a)

Step	Node	PE 0	PE 1	PE 2	PE 3	PE 4	Nodes Dup.
1	n_1	0*	N.C.	N.C.	N.C.	N.C.	NIL
2	n_5	2*	3	N.C.	N.C.	N.C.	NIL
3	n_4	7	2*	N.C.	N.C.	N.C.	n_1
4	n_3	7	6	2*	N.C.	N.C.	n_1
5	n_2	7	6	5	2*	N.C.	n_1
6	n_8	10	11	10*	10	10	n_5
7	n_6	N.C.	N.C.	N.C.	5*	5	NIL
8	n_7	10	10	14	10	8*	n_1, n_2, n_3
9	n_9	N.C.	N.C.	22	21	21	n_6, n_8

(b)

Fig. 6. (a) The schedule generated by the DSH and BTDH algorithms (schedule length = 22), (b) a scheduling trace of the DSH and BTDH algorithms.



(a)

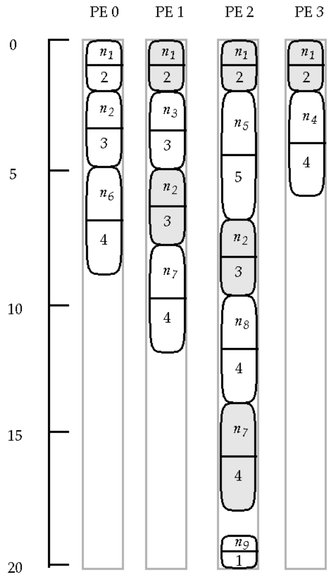
node	*e-values*
n_1	0
n_2	2
n_3	2
n_4	2
n_5	2
n_6	5
n_7	8
n_8	10
n_9	17

(b)

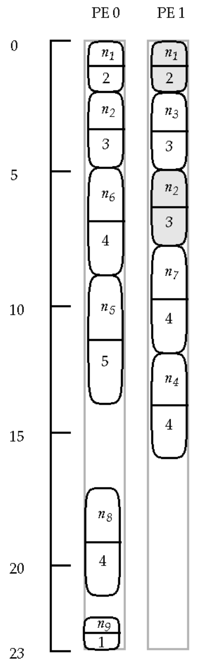
Fig. 7. (a) The schedule generated by the PY algorithm (schedule length = 27), (b) the *e-values* computed by the PY algorithms.

n_1, is selected for scheduling. At the second step, the second CPN (n_7) is selected for scheduling, but its parent nodes n_2 and n_3 are unscheduled. Thus, the procedure *Minimize_start_time* is recursively applied to them. Next, the procedure returns to schedule n_7. Then, the last CPN (n_9) is examined. As most of the ancestor nodes of n_9 are not scheduled, the duplication procedure is also recursively applied to them. Finally, when n_9 is scheduled, only the necessary nodes (n_1, n_2, and n_7) are duplicated. Unlike the other algorithms, the CPFD algorithm does not duplicate the node n_3 when trying to minimize the start time of n_7.

The schedule generated by the ECPFD algorithm is shown in Fig. 8c. The algorithm is given only two processors but still generates a reasonably good schedule. The schedule length generated is about 5 percent longer than those of the other algorithms.



Step	Node	Start times					Nodes
		PE 0	PE 1	PE 2	PE 3	PE 4	Dup.
1	n_1	0*	N.C.	N.C.	N.C.	N.C.	NIL
2	n_2	2*	2	N.C.	N.C.	N.C.	NIL
3	n_3	5	2*	N.C.	N.C.	N.C.	n_1
4	n_7	8	8*	8	N.C.	N.C.	n_2
5	n_6	5*	12	5	N.C.	N.C.	NIL
6	n_5	9	12	2*	N.C.	N.C.	n_1
7	n_4	9	12	7	2*	N.C.	n_1
8	n_8	14	17	10*	11	10	n_2
9	n_9	22	21	19*	N.C.	19	n_7



(a) (b) (c)

Fig. 8. (a) The schedule generated by the CPFD algorithm (schedule length = 20), (b) the scheduling steps of the CPFD algorithm (a node is scheduled to the processor on which the start time is marked by an asterisk and entry with "N.C." indicates the processor is not considered), (c) the schedule genereated by the ECPFD algorithm given only two processors (schedule length = 23.).

5 Performance and Comparison

We implemented the proposed algorithms as well as the four[3] previously reported algorithms on a SUN SPARC Station 2 using suites of regular and irregular task graphs. The regular graphs represent four parallel applications: the parallel Gaussian elimination [23], the mean value analysis [3], the Laplace equation solver [23], and the LU-decomposition [15]. The irregular graphs include the in-tree, out-tree, fork-join, and completely random task graphs [3].

In each graph, the computation costs of the individual nodes were randomly selected from a uniform distribution with the mean equal to the chosen average computation cost. Similarly, the communication costs of the edges were randomly selected from a uniform distribution with the mean equal to the average communication cost. The regular graphs were generated according to their predefined structures. The irregular task graphs were generated in the following manner: Given the number of nodes v, we randomly generated the height (the number of levels) of the graph from a uniform distribution with mean $\sqrt{v}$. At each level, we randomly generated the number of nodes from the same uniform distribution with mean $\sqrt{v}$. The nodes at a level were then randomly connected to nodes at a higher level.

Within each type of graph, we used seven values of CCR: 0.1, 0.5, 1, 1.5, 2, 5, and 10. For each of these values, we generated 10 different graphs of various sizes. For irregular graphs, the number of nodes varied from 50 to 500 with increments of 50. On the other hand, the regular graphs are characterized by the size of their input data matrix. If N is the size of the matrix, the number of nodes is roughly equal to N^2. The size of the matrix was varied from 15 to 24. Thus, for each type of graph structure, 70 graphs were generated, with the total number of graphs corresponding to 560 (eight graph types, seven CCRs, 10 graph sizes).

The performance comparison of the CPFD, LWB, LCTD, DSH, BTDH, and PY algorithm was done in a number of ways. First, the schedules lengths produced by these algorithms were compared with each other by varying graph sizes, graph types, and various values of CCR. Second, we compared the number of times each algorithm produced the best solution. Third, we observed the percentage degradation in performance of an algorithm compared to the best solution. Fourth, a global comparison (each algorithm was compared against the rest of the algorithms) was done. The running times and the number of processors used by each algorithm were also noted. Finally, the effectiveness of the ECPFD algorithm was tested by comparing its performance with that of the CPFD algorithm.

5.1 Comparison of Schedule Lengths

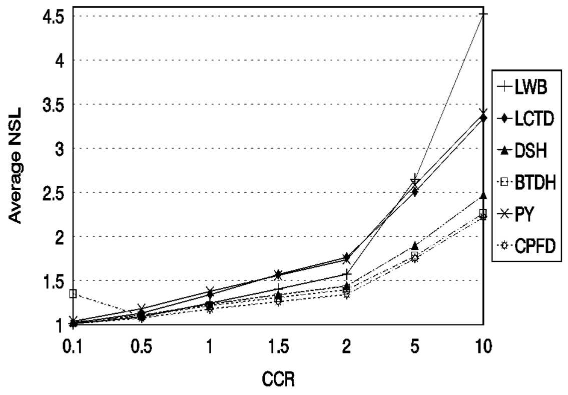
For the first comparison, we present the normalized schedule lengths (NSLs) produced by the six algorithms. An NSL was obtained by dividing the output schedule length by the sum of computation costs on the critical-path.

Table 2 shows the average NSLs produced by each algorithm for each graph type (averaged over seven values of CCR and 10 graph sizes). The right most column shows the average taken across all graph types (average of 560 graphs). These numbers clearly indicate that the CPFD algorithm produces the shortest average schedule length not only across all graphs types but also for each type of graph. The performance of CPFD is followed by that of the BTDH

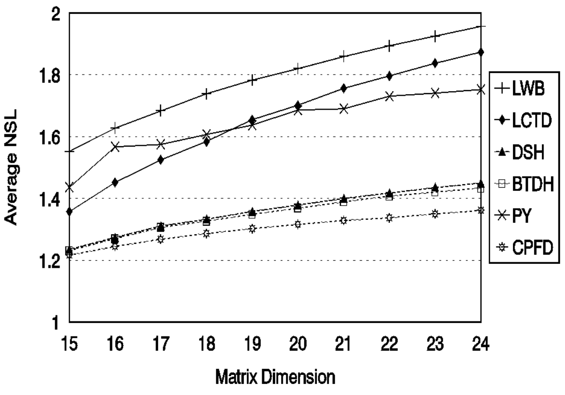
3. We thank Michael Palis for providing us the implementation of the PY algorithm.

TABLE 2
COMPARISON OF AVERAGE NSLS FOR VARIOUS GRAPH TYPES

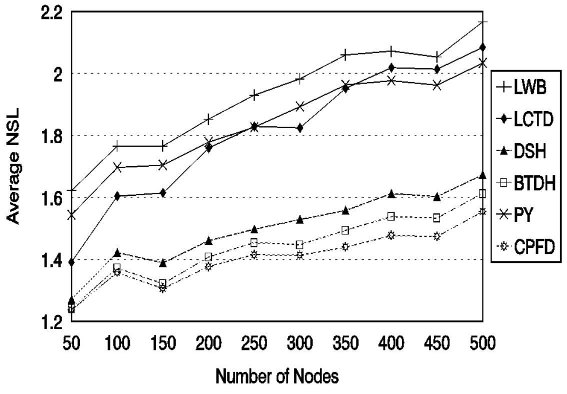
	Graph Types								
Algorithm	Gauss	LU	Laplace	MVA	InTree	OutTree	ForkJoin	Random	Average
LWB	2.44	1.33	1.61	2.23	2.50	1.00	2.20	2.13	1.93
LCTD	1.75	1.34	1.95	2.01	2.10	1.49	1.97	1.86	1.81
DSH	1.30	1.21	1.53	1.60	1.84	1.23	1.66	1.65	1.50
BTDH	1.29	1.21	1.53	1.56	1.80	1.00	1.59	1.56	1.44
PY	1.86	1.50	1.69	1.93	2.12	1.24	2.29	2.07	1.84
CPFD	1.28	1.16	1.46	1.53	1.77	1.00	1.54	1.51	1.41



(a)



(b)



(c)

Fig. 9. Average NSL of the six algorithms for various CCRs of all graphs for various sizes of regular graphs and random graphs.

algorithm. There is a little difference between the performance of DSH and BTDH. The LWB algorithm exhibits large variations in its performance. It performs well for LU-decomposition and out-trees but does not perform well on other graphs. Based on this comparison, these algorithm are ranked in the following order: CPFD, BTDH, DSH, LCTD, PY, and LWB.

Fig. 9a shows the NSLs (averaged over all graph sizes and graph types) of each algorithm against various values of CCR. From this figure, we can observe that all algorithm are very sensitive to the value of CCR. This is because a larger value of CCR can have more impact (positive or negative) on the schedule lengths. A large value of CCR thus tests the capabilities of an algorithm more robustly, and we can also notice that the differences between the performance of various algorithms become more significant with larger value of CCR. The relative performance of these algorithms is, however, consistent with our earlier conclusion.

Fig. 9b and Fig. 9c show the NSLs yielded by each algorithm against various graph types (averaged across graph types and CCRs) for regular and irregular graphs, respectively. CPFD is again consistently better than all of the other algorithms. The size of the graph, both for regular and irregular types, has no bearing on this observation.

Fig. 10 contains six Kiviat graphs, each of which shows the number of cases in which an algorithm generated the best solution and the number of cases the percentage degradation in schedule length from the best solution is within a certain range. That is, for each test case, we compared the schedule length produced by an

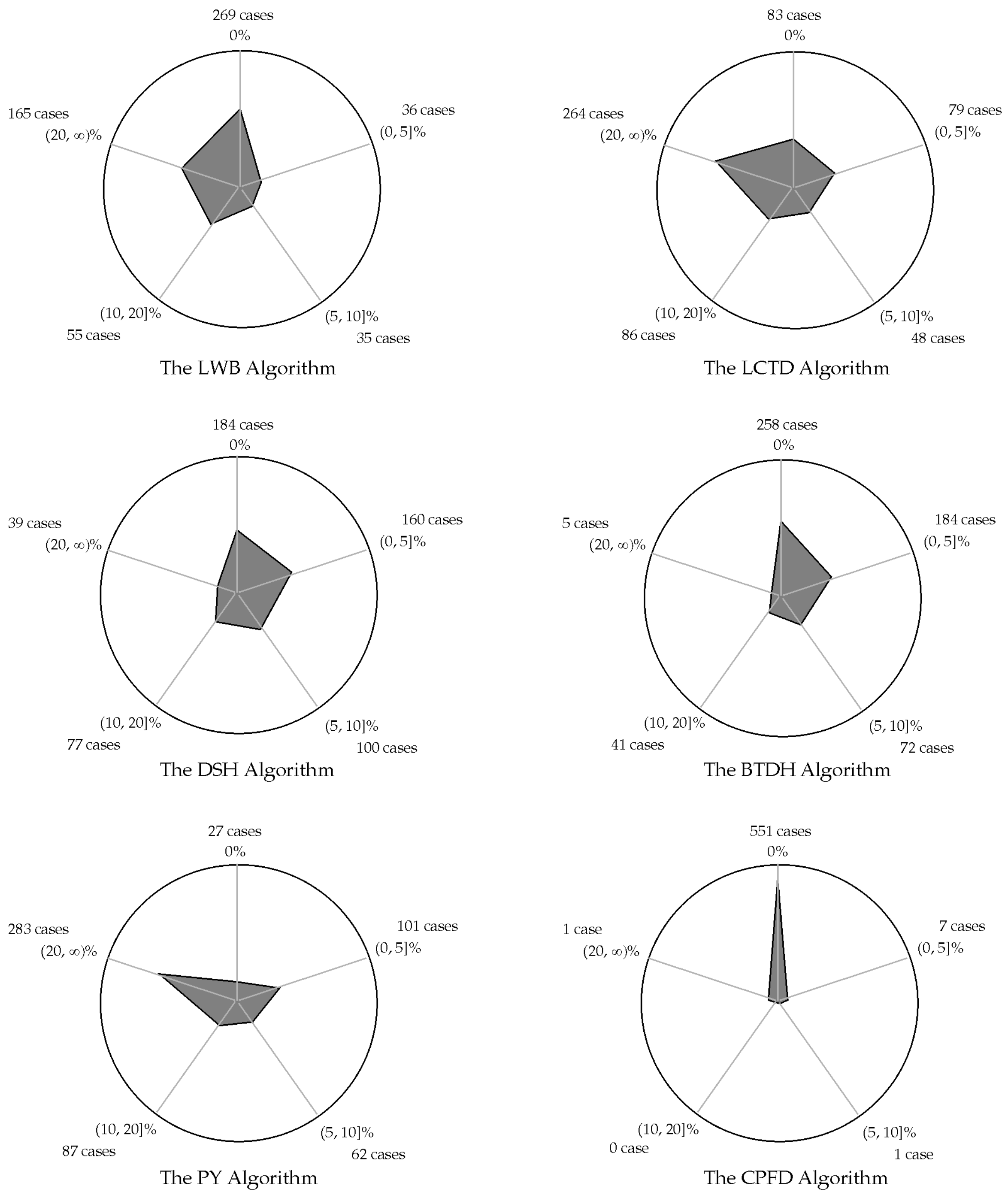


Fig. 10. Kiviat graphs showing the performance of the six algorithms in terms of the number of best schedule lengths generated and the number of cases in which the schedule length percentage degradations from the best solutions are within various ranges for all the task graphs.

algorithm with the best solution out of all the algorithms and measured the amount of degradation. As such, each Kiviat graph has five polar axes: the number of best solutions generated by the algorithm (0 percent) degradation), the number of cases when the degradation is less than or equal to 5 percent, the number of cases when the degradation is between 5 to 10 percent, one indicates the number of cases when the degradation is between 10 to 20 percent, and the number of cases when the degradation is more than 20 percent. The full scale of each axis is 560, which is the total number of test cases.

Clearly, if the shaded region in a Kiviat graph closely surrounds the 0 percent-axis, the corresponding algorithm generates the best solutions in most of the cases. Fig. 10 indicates that the CPFD algorithm is the only algorithm with such a consistent performance. Indeed, the CPFD algorithm produced the best solutions 551 times out of 560 trials. On the other hand, out of nine cases in which the CPFD algorithm failed to produce the best solution, its performance degradation is less than 5 percent in seven cases and is more than 20 percent in one case only. In contrast, the performance degradation of the other

TABLE 3
COMPARISON OF THE NUMBER OF BEST SCHEDULE LENGTHS GENERATED AND THE NUMBER OF CASES IN WHICH THE PERCENTAGE DEGRADATIONS FROM THE BEST SCHEDULE LENGTHS ARE WITHIN CERTAIN INTERVALS FOR VARIOUS GRAPH TYPES

% degradations from the best solutions	Algorithm	Graph Types								Total
		Gauss	LU	Laplace	MVA	InTree	OutTree	ForkJoin	Random	
0%	LWB	8	50	50	23	26	70	21	21	269
	LCTD	1	20	0	0	29	20	2	11	83
	DSH	35	33	19	10	39	23	7	18	184
	BTDH	35	33	19	15	42	70	19	25	258
	PY	1	0	1	0	8	14	1	2	27
	CPFD	70	70	68	70	68	70	70	65	551
(0, 5]%	LWB	3	0	0	7	8	0	11	7	36
	LCTD	10	10	10	10	8	5	15	11	79
	DSH	24	14	18	33	16	6	32	17	160
	BTDH	24	14	18	46	19	0	42	21	184
	PY	2	10	20	15	19	11	13	11	101
	CPFD	0	0	2	0	2	0	0	3	7
(5, 10]%	LWB	3	0	6	5	6	0	7	8	35
	LCTD	3	1	11	8	8	6	5	6	48
	DSH	11	5	13	23	9	8	15	16	100
	BTDH	11	5	13	9	7	0	7	20	72
	PY	8	3	13	9	7	6	10	6	62
	CPFD	0	0	0	0	0	0	0	1	1
(10, 20]%	LWB	6	10	4	11	5	0	9	10	55
	LCTD	10	15	15	12	5	10	7	12	86
	DSH	0	14	19	4	5	11	12	12	77
	BTDH	0	14	19	0	2	0	2	4	41
	PY	1	9	18	9	16	13	7	14	87
	CPFD	0	0	0	0	0	0	0	0	0
(20, ∞)%	LWB	50	10	10	24	25	0	22	24	165
	LCTD	46	24	34	40	20	29	41	30	264
	DSH	0	4	1	0	1	22	4	7	39
	BTDH	0	4	1	0	0	0	0	0	5
	PY	58	48	18	37	20	26	39	37	283
	CPFD	0	0	0	0	0	0	0	1	1

algorithms can be seen in all ranges. The performance degradation of the LCTD and PY algorithms in the range of 20 percent or higher is more frequent compared to the rest of the algorithms. The performance of the BTDH and DSH algorithm is better than the LWB, LCTD, and PY algorithms. The LWB algorithm, as noted earlier, exhibits large fluctuations in performance.

Next, we show the number of best solutions and the number of cases with nonzero percentage degradations of each algorithm across all the graph types and across all values of CCR in Table 3 and Table 4, respectively. From Table 3, we can see that the CPFD algorithm could not produce the best solution two times for Laplace solver graphs, two times for in-tree graphs, and five times for random graphs. The performance of other algorithms varies considerably for different graph types. From Table 4, we observe that the LWB and LCTD algorithms perform well only with low values of CCR. The other algorithms appear to be insensitive to the value of CCR.

Fig. 11 and Fig. 12 depict the Kiviat graphs showing the average percentage degradations from the best solutions across all the graph types and values of CCR, respectively. In these graphs, the full scale of each axis represents a degradation of 85 percent. It should be noted that a smaller shaded region implies a better performance. The CPFD algorithm is thus the best algorithm in that its Kiviat graph has the smallest shaded region and its overall performance degradation from the best solutions is only 1 percent. Another observation from Fig. 11 is that the LWB, BTDH, and CPFD algorithms generate the best solutions for all the out-tree task graphs. Fig. 12 reveals that the LWB, LCTD, and PY algorithms are sensitive to CCR in that their performance degrades as CCR increases.

5.2 A Global Comparison

In the global one-to-one comparison among the six algorithms, we observe the number of times (out of 560 trials) an algorithm yielded a better, worse, or the same schedule length compared to each of the other five algorithms. This comparison is shown in Fig. 13. A box in this figure compares one of the algorithms on the left with one of the algorithms on the top. A box has three numbers indicating the number of times the algorithm on the left performed better (>), worse (<), or the same (=) compared to the algorithm on the top. For instance, the CPFD algorithm compared with the PY algorithm yielded better solutions 533 times, worse solutions two times, and the same solution 25 times.

An algorithm's performance compared to the rest of the five algorithms combined is shown in the box on the right side. The CPFD outperformed all other algorithms in 1,975 cases, but was outperformed in 14 cases only. Using this figure, while we can observe the performance

TABLE 4
COMPARISON OF THE NUMBER OF BEST SCHEDULE LENGTHS GENERATED AND THE NUMBER OF CASES IN WHICH THE PERCENTAGE DEGRADATIONS FROM THE BEST SCHEDULE LENGTHS ARE WITHIN CERTAIN INTERVALS FOR VARIOUS CCRS

% degradations from the best solutions	Algorithm	CCRs						
		0.1	0.5	1.0	1.5	2.0	5.0	10.0
0%	LWB	78	67	42	10	32	30	10
	LCTD	38	23	8	4	7	2	1
	DSH	45	33	17	25	17	19	28
	BTDH	44	37	29	46	27	29	46
	PY	2	2	3	10	4	3	3
	CPFD	80	80	80	77	80	79	75
(0, 5]%	LWB	2	4	20	9	1	0	0
	LCTD	42	16	17	2	1	0	1
	DSH	26	32	33	27	14	14	14
	BTDH	20	32	31	26	26	25	24
	PY	69	22	3	2	1	1	3
	CPFD	0	0	0	3	0	0	4
(5, 10]%	LWB	0	3	8	10	8	6	0
	LCTD	0	29	7	6	2	3	1
	DSH	3	10	18	18	24	16	11
	BTDH	11	7	14	16	11	7	6
	PY	9	26	18	7	2	0	0
	CPFD	0	0	0	0	0	1	0
(10, 20]%	LWB	0	6	0	18	17	14	0
	LCTD	0	12	26	21	12	9	6
	DSH	5	5	11	15	15	13	13
	BTDH	4	4	6	11	10	2	4
	PY	0	21	25	18	14	4	5
	CPFD	0	0	0	0	0	0	0
(20, ∞)%	LWB	0	0	10	11	24	50	70
	LCTD	0	0	22	44	63	67	68
	DSH	1	0	1	3	8	9	17
	BTDH	1	0	0	0	4	0	0
	PY	0	9	31	49	60	72	62
	CPFD	0	0	0	0	0	0	1

of any algorithm, we can also observe that the differences between the performance of other algorithms are not as large as the difference between the performance of CFPD algorithm and the rest of the algorithms. This indicates that the performance improvement of the CPFD algorithm is not marginal.

5.3 Number of Processors Used

All six algorithms described above work under the assumption of the availability of unbounded number of processors, but each algorithm has its own philosophy of utilizing processors. We measured the number of processors used by each algorithm for different graph sizes. These numbers are plotted in Fig. 14a and Fig. 14b for regular and irregular graphs, respectively. The differences among these numbers for various algorithms are large. The LWB algorithm, for instance, uses a very large number of processors while the DSH algorithms uses fewer processors.

5.4 Comparison of Running Times

Here we include the measured running times of all algorithms running on a SUN SPARC workstation. These times are plotted in Fig. 15a and Fig. 15b for regular and irregular graphs, respectively. The complexities of these algorithms mentioned earlier concur with the measured timings. The LWB and PY algorithms are faster than the rest of the algorithms. The timings of the CPFD algorithm are slightly larger than those of the LCTD algorithm. However, since the main objective of our algorithm is minimization of the schedule length and the scheduling is done off-line, a slightly longer time in generating a considerably improved solution should be acceptable. The time to schedule a very large graph (4.3 seconds for 500 node task graph) is still reasonable. The timings of the BTDH and DSH are also similar, with the former being slightly slower than the latter.

5.5 Comparison Using Previous Benchmarks

We also applied the six algorithms to a number of example task graphs used by various researchers. The results summarized in Table 5 indicate that only the CPFD algorithm generated the shortest schedules for all the graphs. These results also indicate that despite the very small sizes of the example graphs, the performance of all the five previously reported algorithms varied significantly.

5.6 The Performance of the ECPFD Algorithm

Finding a suitable processor for duplication under the assumption of the availability of unbounded number of

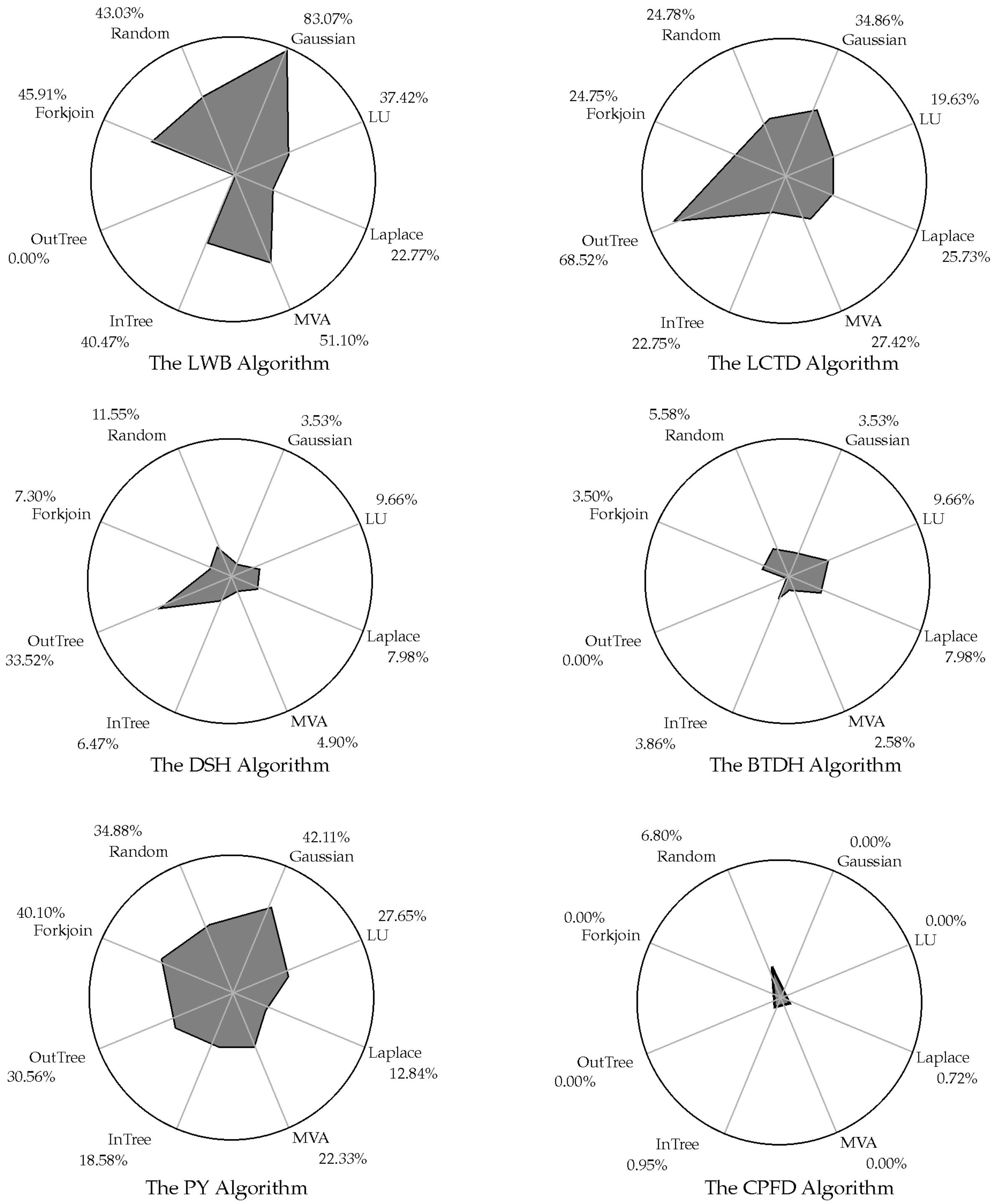


Fig. 11. Kiviat graphs showing the average percentage degradations from the best schedule lengths of the six algorithms for various types of task graphs.

processors is easier for generating shorter solutions. On the other hand, unlimited number of processors may not always be true. While in our study, the CPFD algorithm is more comprehensively evaluated, the ECPFD algorithm working under the assumption of a bounded number of processors is more efficient and useful. In this section, we compare the ECPFD algorithm with the CPFD algorithm and show that by using considerably less number of processors, the ECPFD algorithm generates solutions which are close to the CPFD algorithm. The comparison was done by first measuring the number of processors used by CPFD and then giving only 50 percent of this number to ECPFD as the input.

Fig. 16 shows the comparison of the schedule lengths generated by the CPFD and ECPFD algorithms against various parameters (graph type, CCR, and graph size for regular and irregular graphs). The schedule lengths generated by the ECPFD algorithm are only marginally longer compared to those generated by the CPFD algorithm. The magnitude of this difference is between 1.5 to

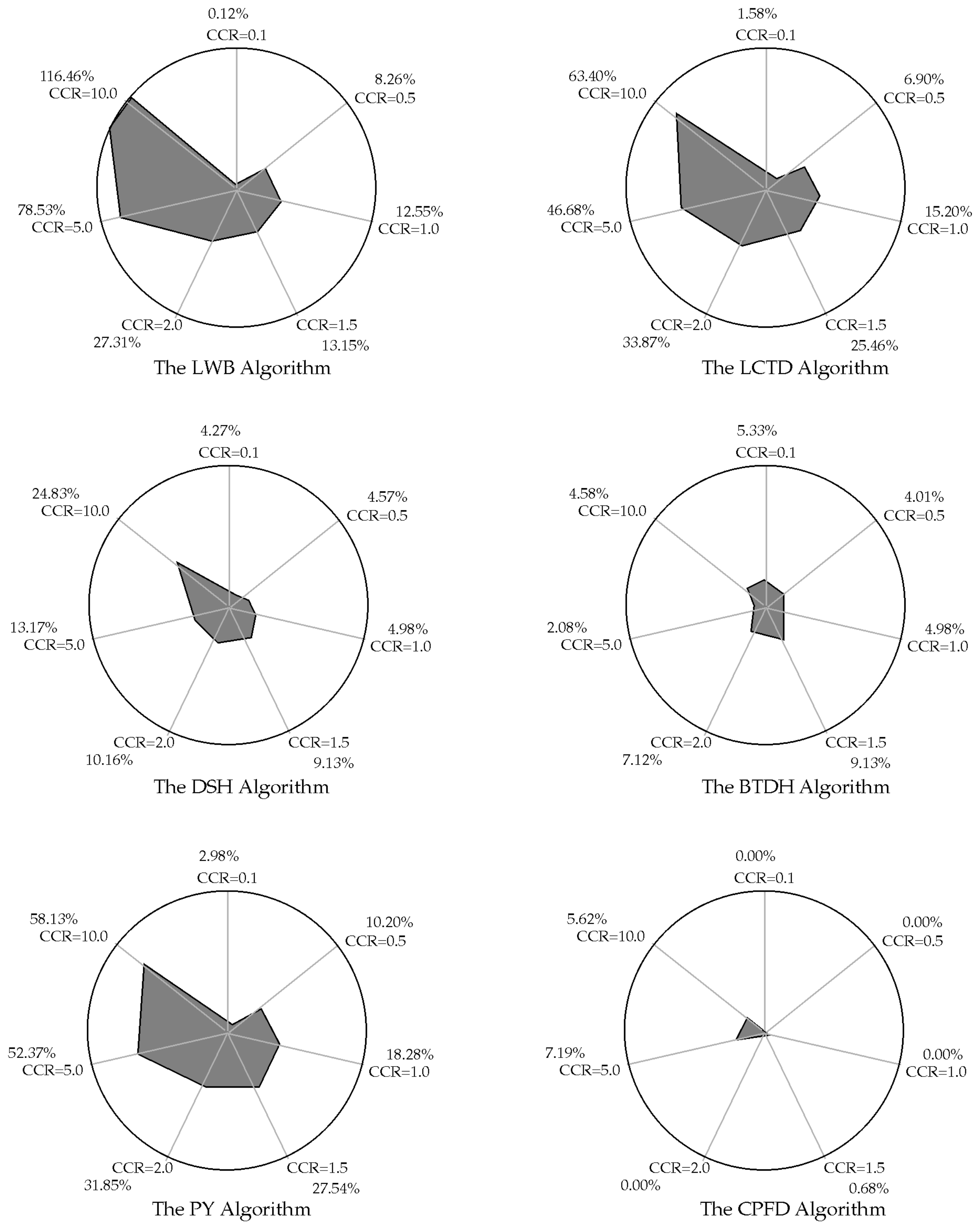


Fig. 12. Kiviat graphs showing the average percentage degradations from the best schedule lengths of the six algorithms for various CCRs.

5 percent for regular graphs and 5 to 11 percent for irregular graphs. This difference, we believe, is acceptable given that ECPFD used only 50 percent processors. The results using even smaller number of processors are also found to be useful but are not reported here due to lack of space.

6 Conclusions

In this paper, we have discussed the problem of using duplication in scheduling parallel programs represented by directed tasks graphs with arbitrary computation and communication costs. We have analyzed and evaluated five previously proposed algorithms and outlined some principles for generating better solutions using duplication. The technique used in our proposed algorithms is to systematically decompose the task graph into CPN, IBN, and OBN bindings. These bindings help in identifying the relative importance of nodes and then enable them to start at their earliest possible start times according. In our experimental study, the CPFD algorithm consistently outperforms all of

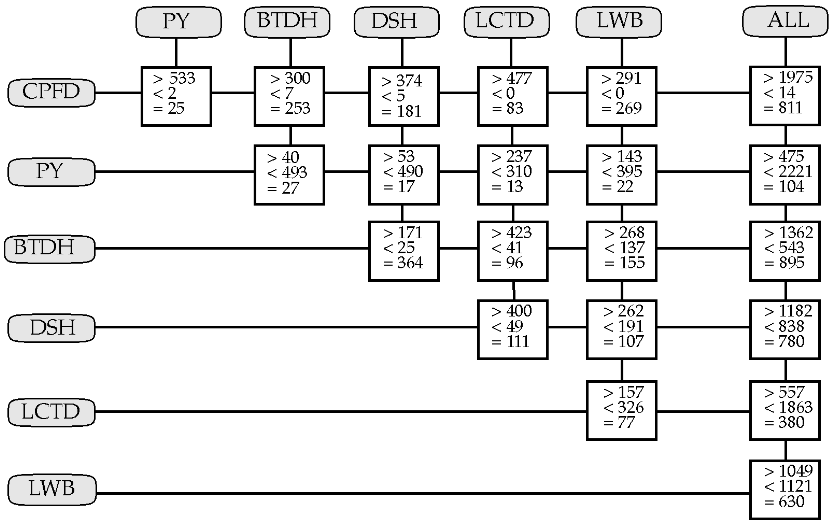


Fig. 13. A global comparison of the six scheduling algorithms in terms of better, worse, and equal performance.

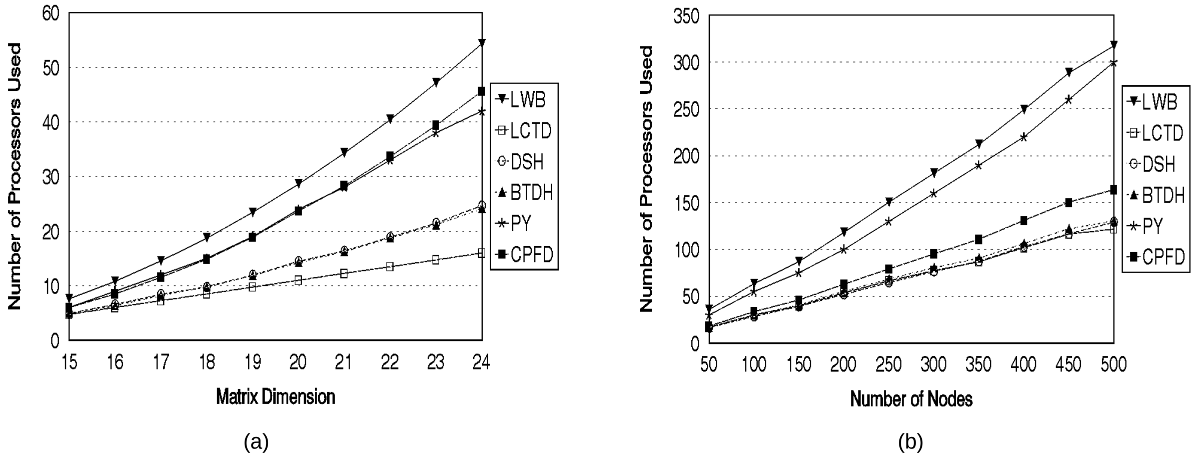


Fig. 14. The average number of processors used by the six scheduling algorithms for (a) regular task graphs of various matrix dimensions and (b) irregular task graphs of various sizes.

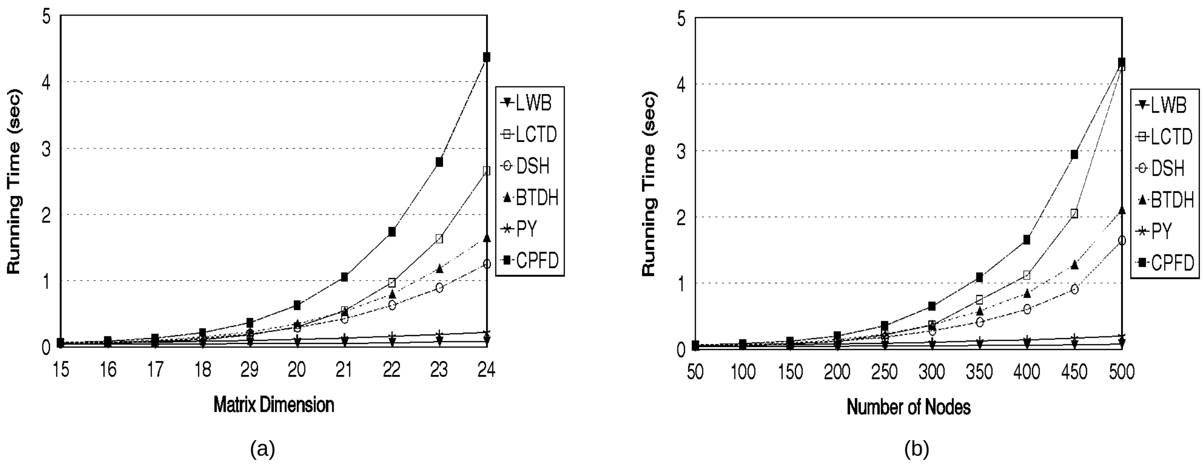


Fig. 15. The running time of the six scheduling algorithms on a SPARC Station 2 for (a) regular task graphs of various matrix dimensions and (b) irregular task graphs of various sizes.

the previous algorithms. The ECPFD algorithm controls the degree of duplication according to the number of available processors. The quality of solution produced by CPFD is slightly better than ECPFD. However, ECPFD is more efficient as it uses fewer number of processors.

Appendix

Proof of Theorem 1. It suffices to prove the statement: The CPFD algorithm schedules every node of an out-tree to start at its absolute earliest start time, which equals

TABLE 5
SCHEDULE LENGTHS GENERATED BY THE SIX ALGORITHMS FOR SOME EXAMPLE TASK GRAPHS

Source of task graph	BTDH	DSH	LCTD	LWB	PY	CPFD
Al-Maasarani [1] (a 16-node graph)	40	44	42	41	49	40
Al-Mouhamed [2] (a 17-node graph)	35	35	37	37	39	35
Shirazi, Chen and Marquis [20] (a 11-node graph)	25	25	25	36	29	25
Colin and Chretienne [5] (a 9-node graph)	13	13	15	12	14	12
Gerasoulis and Yang [7] (a 7-node graph)	16	16	20	17	26	16
Kruatrachue and Lewis [12] (a 15-node graph)	361	361	462	761	369	361
McCreary and Gill [16] (a 9-node graph)	127	127	180	130	219	127
Chung and Ranka [4] (a 11-node graph)	32	32	32	39	36	32
Wu and Gajski [23] (a 18-node graph)	320	320	390	390	420	320
Yang and Gerasoulis[24] (a 7-node graph)	16	16	18	18	22	16

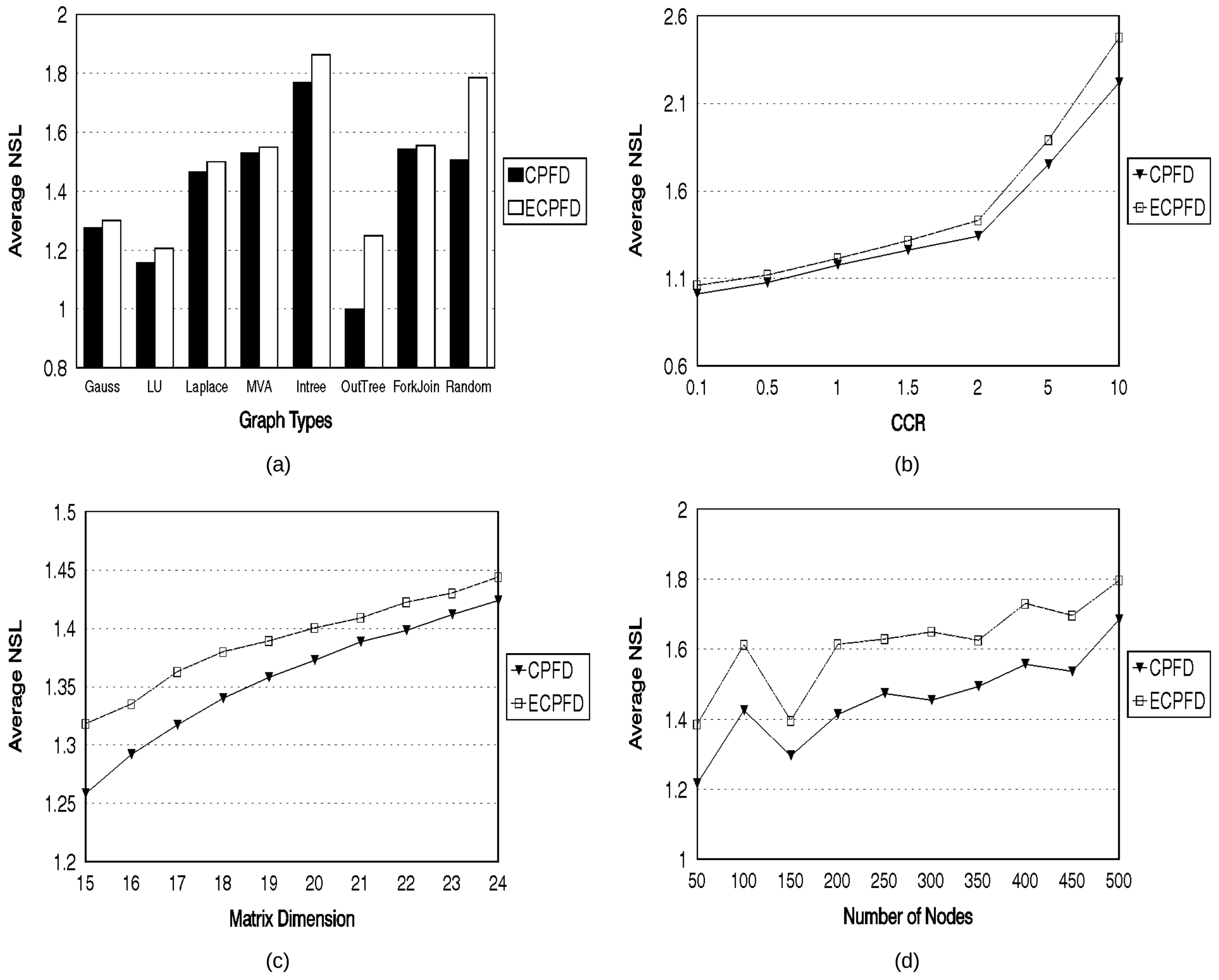
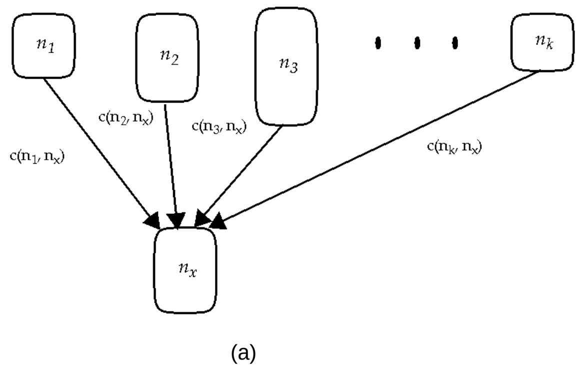


Fig. 16. Comparisons of average NSLs of CPFD against ECPFD (with 50 percent used by ECPFD) for various (a) graphs types, (b) CCRs, (c) matrix dimensions of regular graphs, (d) sizes of irregular graphs.

to the sum of computation costs from the root to the parent of the node. We prove this by induction on the depth of a node. For the root node, the statement obviously holds. For a node n_x at depth k, be it a CPN or an OBN (note that there is no IBN in an out-tree), the CPFD algorithm will recursively consider its ancestors before considering to schedule it. By the induction assumption, n_x's only parent is scheduled to start at the earliest time, which is the sum of computation costs from the root up to the parent of n_x's parent. Since n_x has only one parent, the start time minimization procedure of the CPFD algorithm will lead to scheduling n_x to the same processor as its parent. Thus, n_x also starts at the absolute earliest time. The theorem is proved. □

PROOF OF THEOREM 2. Consider an in-tree graph with height one as shown in Fig. 17a. Suppose that:



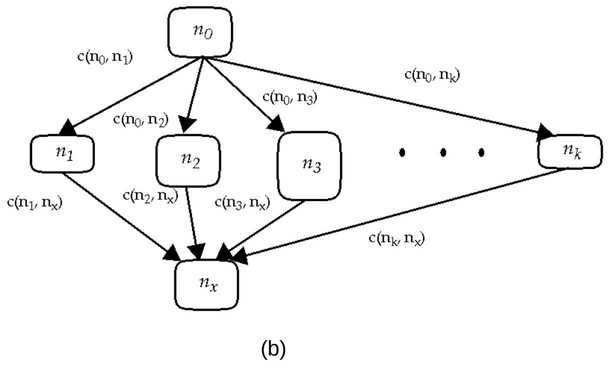


Fig. 17. (a) An in-tree graph with height equal to one, (b) a unit-lobe fork-join graph.

$$w(n_1) + c(n_1, n_x) \geq w(n_2) + c(n_2, n_x) \geq \ldots \geq w(n_k) + c(n_k, n_x).$$

We will prove that the CPFD algorithm generates an optimal schedule whose length is equal to

$$\max\left\{\sum_{i=1}^{j} w(n_i),\, w(n_{j+1}) + c(n_{j+1}, n_x)\right\} + w(n_x),$$

where

$$\sum_{i=1}^{j} w(n_i) \leq w(n_j) + c(n_j, n_x)$$

and

$$\sum_{i=1}^{j+1} w(n_i) > w(n_{j+1}) + c(n_{j+1}, n_x).$$

According to the CPFD algorithm, all the entry nodes are scheduled to an empty processor so that they start at time 0. When n_x is examined for scheduling, the node n_1 will be considered for duplication because it is the first VIP. After duplicating n_1, the start time of n_x will improve because the communication from n_1 is zeroed. The start time is now constrained by the data from n_2. Thus, n_2 becomes the new VIP and is also duplicated. This process is repeated until the VIP n_{j+1} is considered. At that point, the start time of n_x will increase if n_{j+1} is duplicated so that no more VIP will be duplicated and the schedule length will be the one stated above. This schedule length is optimal because the node n_x cannot start any earlier. □

PROOF OF THEOREM 3. We first define the notion of a *lobe* for a fork-join task graph. A lobe consists of a fork node, its children, and the join node. Thus, for example, in Fig. 2c, the nodes $\{n_1, n_2, n_3, n_4, n_5\}$ constitute the first lobe and the nodes $\{n_5, n_6, n_7, n_8\}$ constitute the second. To prove the theorem, it suffices to prove that the CPFD algorithm schedules all the nodes in a fork-join to start at the absolute earliest start time. We prove this by induction on the number of lobes. Consider the case when there is only one lobe, as shown in Fig. 17b. Let us assume that the following holds:

$$w(n_1) + c(n_1, n_x) \geq w(n_2) + c(n_2, n_x) \geq \ldots \geq w(n_k) + c(n_k, n_x).$$

We will prove that the CPFD algorithm generates an optimal schedule whose length is equal to:

$$w(n_0) + \max\left\{\sum_{i=1}^{j} w(n_i),\, w(n_{j+1}) + c(n_{j+1}, n_x)\right\} + w(n_x),$$

where

$$\sum_{i=1}^{j} w(n_i) \leq w(n_j) + c(n_j, n_x)$$

and

$$\sum_{i=1}^{j+1} w(n_i) > w(n_{j+1}) + c(n_{j+1}, n_x).$$

According to the CPFD algorithm, the entry node n_0 is scheduled to an empty processor so that it starts at time 0. Then, all of its children nodes will be scheduled to distinct processors so that all of them can start when n_0 finishes. When n_x is examined for scheduling, the node n_1 will be considered for duplication because it is the first VIP. As the procedure *Minimize_start_time* is recursively applied to it, the node n_0 will also be duplicated. After duplicating n_1 and n_0, the start time of n_x will improve because the communication from n_1 is zeroed. The start time is now constrained by the data from n_2. Thus, n_2 becomes the new VIP and is also duplicated. This process will repeat until the VIP n_{j+1} is considered. At that point, the start time of n_x will increase if n_{j+1} is duplicated so that no more VIP will be duplicated and the schedule length will be the one stated above. This schedule length is optimal because the node n_x cannot start any earlier. Thus, the theorem holds for the case of a unit-lobe fork- join. Suppose the fork-join task graph has k lobes as shown in Fig. 18. Let us assume that the following holds:

$$w(n_s) + c(n_s, n_z) \geq \ldots \geq w(n_t) + c(n_t, n_z).$$

By the induction assumption, the node n_y and all its ancestors will be scheduled to start at the absolute earliest start times. When the nodes n_s to n_t are considered, the start time minimization procedure of the algorithm will duplicate n_y together with its ancestors, which have been duplicated on the same processor as n_y when n_y is scheduled for the first

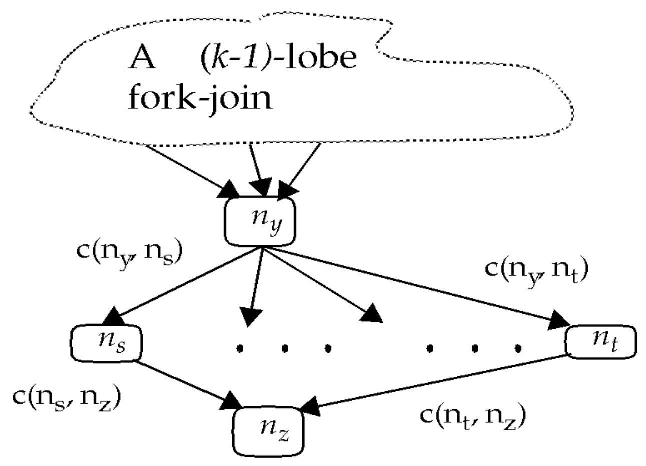


Fig. 18. A *k*-lobe fork-join graph.

time, so that n_y and, in turn, n_s, can start at the earliest time. Since n_y cannot start any earlier, the nodes n_s to n_t are also scheduled to start at the absolute earliest times. Note that each of the nodes n_s to n_t will be scheduled using a distinct processor with the same set of ancestors duplicated because, otherwise, the start time of n_y will not be minimum. Call this set of ancestors the set *A*. When n_z is considered for scheduling, n_s will be examined for duplication first as it is the VIP. As the procedure *Minimize_start_time* is recursively applied to n_s, all nodes in *A* will also be duplicated. Using similar argument as in the case of a single lobe fork-join, the parents of n_z will be duplicated up to a point when the cumulative computation costs is greater than the communication time from the next parent. In this way the start time of n_z is also the absolute minimum. Thus, the theorem is proved. □

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their helpful comments. Preliminary versions of portions of this paper were presented at the 1994 International Conference on Parallel Processing and the Sixth IEEE Symposium on Parallel and Distributed Processing. This research was supported by a grant from the Hong Kong Research Grants Council under contract number HKUST 734/96E.

REFERENCES

[1] A. Al-Maasarani, "Priority-Based Scheduling and Evaluation of Precedence Graphs With Communication Times," MS thesis, King Fahd Univ. of Petroleum and Minerals, Saudi Arabia, 1993.

[2] M.A. Al-Mouhamed, "Lower Bound on the Number of Processors and Time for Scheduling Precedence Graphs With Communication Costs," *IEEE Trans. Software Eng.*, vol. 16, no. 12, pp. 1,390-1,401, Dec. 1990.

[3] 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 '92*, pp. 683-691, Nov. 1992.

[4] Y.C. Chung and S. Ranka, "Application and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed-Memory Multiprocessors," *Proc. Supercomputing '92*, pp. 512-521, Nov. 1992.

[5] J.Y. Colin and P. Chretienne, "C.P.M. Scheduling With Small Computation Delays and Task Duplication," *Operations Research*, pp. 680-684, 1991.

[6] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[7] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors," *J. Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276-291, Dec. 1992.

[8] 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.

[9] 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.

[10] 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.

[11] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. Computers*, vol. 33, no. 11, pp. 1,023-1,029, Nov. 1984.

[12] B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, pp. 23-32, Jan. 1988.

[13] Y.-K. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs onto Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 5, pp. 506-521, May 1996.

[14] T.G. Lewis and H. El-Rewini, *Introduction to Parallel Computing*. New York: Prentice Hall, 1992.

[15] 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.

[16] 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.

[17] C. Papadimitriou and M. Yannakakis, "Toward an Architecture Independent Analysis of Parallel Algorithms," *SIAM J. Computing*, vol. 19, pp. 322-328, 1990.

[18] C.D. Polychronopoulos and D.J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,425-1,439, Dec. 1987.

[19] 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.

[20] B. Shirazi, H. Chen, and J. Marquis, "Comparative Study of Task Duplication Static Scheduling versus Clustering and Non-Clustering Techniques," *Concurrency: Practice and Experience*, vol. 7, no. 5, pp. 371-390, Aug. 1995.

[21] B. Shirazi, M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Scheduling," *J. Parallel and Distributed Computing*, vol. 10, pp. 222-232, 1990.

[22] 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. 75-87, Feb. 1993.

[23] 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.

[24] 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, pp. 951-967, Sept. 1994.

Ishfaq Ahmad received a BSc degree in electrical engineering from the University of Engineering and Technology, Lahore, Pakistan, in 1985. He received his MS degree in computer engineering and PhD degree in computer science, both from Syracuse University, in 1987 and 1992, respectively. Currently, he is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology. His research interests are in the areas of parallel programming tools, scheduling and mapping algorithms for scalable architectures, video technology, and interactive multimedia systems. He has published extensively in these areas. He has received numerous research and teaching awards, including the Best Student Paper Award at Supercomputing '90 and Supercomputing '91, and the Teaching Excellence Award by the School of Engineering at the Hong Kong University of Science and Technology. Dr. Ahmad has served on the committees of various international conferences, has been a guest editor for two special issues of *Concurrency: Practice and Experience* related to resource management, and is co-guest-editing a forthcoming special issue of the *Journal of Parallel and Distributed Computing* on the topic of software support for distributed computing. He also serves on the editorial board of Cluster Computing. He is a member of the IEEE and the IEEE Computer Society.

Yu-Kwong Kwok received his BSc degree in computer engineering from the University of Hong Kong in 1991, and the MPhil and PhD degrees in computer science from the Hong Kong University of Science and Technology in 1994 and 1997, respectively. Currently, he is an assistant professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong. Before joining the University of Hong Kong, he was a visiting scholar in the Parallel Processing Laboratory of the School of Electrical and Computer Engineering at Purdue University for one year. His research interests include software support for parallel and distributed processing, heterogeneous computing, and multimedia systems. He is a member of the IEEE, the IEEE Computer Society, and the ACM.