A New Approach To Scheduling Parallel Programs Using Task Duplication

Ishfaq Ahmad and Yu-Kwong Kwok
Department of Computer Science
Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Abstract[1]
In this paper, we explore the problem of scheduling parallel programs using task duplication for message-passing multicomputers. Task duplication means scheduling a parallel program by redundantly executing some of the tasks on which other tasks of the program critically depend. This can reduce the start times of tasks waiting for messages from tasks residing in other processors. There have been a few scheduling algorithms using task duplication. We discuss two such previously reported algorithms and describe their differences, limitations and suitability for different environments. A new algorithm is proposed which outperforms both of these algorithms, and is more efficient for low as well as high values of communication-to-computation ratios. The algorithm takes into account arbitrary computation and communication costs. All three algorithms are tested by scheduling some of the commonly encountered graph structures.

1 Introduction

Despite great architectural advances, the communication overhead in message-passing parallel computers in general and in networked distributed systems in particular remains an inevitable penalty. Due to this penalty, the speedup of a parallel program may be limited or may not scale very well with the size of the system. The interprocessor communication overhead occurs when two tasks of a parallel program assigned to different processors have dependencies and they need to exchange data among them [9]. Task duplication is one way of reducing the interprocessor communication overhead which in turn can improve the total execution time [6]. Task duplication means scheduling a parallel program by redundantly allocating some of its tasks on which other tasks critically depend. This reduces the start times of waiting tasks which can eventually improve the overall execution time of the whole program. Duplication based scheduling can be particularly useful for systems with high communication overhead such as a network of workstations.

To effectively run a parallel program on an architecture, the program needs to be scheduled in an efficient fashion. The scheduling problem can be described as an allocation of a set of tasks onto a set of processors, such that the total schedule length in terms of time is minimized. It is a well-known fact that the scheduling problem in its many variants is NP-complete [2] and most of the solutions are based on heuristics [4], [9], [10]. The complexity of the algorithm and the quality of the solution largely depend on the task graph structure and the target machine model. The algorithm's complexity should be within practical limits and it should be scalable in that it should still generate a good solution if the size of the problem and system is increased. Even with an efficient scheduling algorithm, it may happen that some processors are idle during different time periods because the tasks assigned to them are waiting to receive some data from the tasks assigned to some other processors. If these idle time slots can be utilized effectively by identifying the critical tasks and redundantly allocating them in these slots, the execution time of the parallel program can be further reduced. However, using duplication makes the scheduling problem more difficult. The scheduling algorithm not only needs to observe the precedence constraints among tasks but also needs to recognize which tasks to duplicate and how to fit them in the idle time slots.

This paper is organized as follows. In Section 2, we first describe the problem statement and present some definitions used in our study. We also discuss two previously proposed scheduling algorithms in the same section. In Section 3, we first outline the basic principles used in the design of our algorithm. We also describe our proposed algorithm. Section 4 contains the experimental results and performance comparisons. The last section concludes this paper.

2 Problem Statement and Related Algorithms

In this section, we describe the problem statement through an introduction of a number of terminology commonly used for the scheduling problem. We also present some discussion on using duplication in the scheduling problem. Two previously reported scheduling algorithms using duplication, as well as their characteristics, are discussed at the end of this section.

A parallel program can be represented by a directed acyclic graph in which each node, denoted by n_i, represents a task. The amount of computation required in a task is called the *computation cost* and is denoted by $w(n_i)$. The edges in the parallel program graph correspond to the communication messages and precedence constraints among the tasks. A number is associated with each edge to denote the amount of communication data from a task to another. This number is called the *communication cost* 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. A node which does not have parent node is called an *entry* node whereas a node which does not have child node is called an *exit* node. Clearly, the values of $w(n_i)$ and c_{ij} depend not only on the parallel program but also on the parameters of the underlying system. For example, even a small amount of communication data when routed over a very slow network can result in a very high value of c_{ij}. 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. The communication cost among two nodes assigned to the same processor is assumed to be zero. If n_i is scheduled to

1. This research was supported by Hong Kong's RGC grant under contract number HKUST179/93E.

processor J, $ST(n_i, J)$ and $FT(n_i, J)$ denote the start time and finish time of n_i on processor J, respectively. It should be noted that $FT(n_i, J) = ST(n_i, J) + w(n_i)$. After all nodes have been scheduled, the schedule length is defined as $max_i\{FT(n_i, J)\}$ across all processors.

2.1 Fundamentals

A node cannot start execution before it gathers all of the messages from its parent nodes. Thus, it is not possible to determine the start time of a node before determining the start times of its parent nodes. This implies that a *node cannot be scheduled until all of its parent nodes have been scheduled.* When all the parent nodes of a node have been scheduled, there is a constraint on its start time which is due to the communication edges from its parent nodes. This is explained by the following definition.

Definition 1: *The communication-constrained earliest start time, denoted by $CEST(n_i, J)$, of a node n_i on a processor J is defined as*

$$max_{1<k<p}\{FT(n_{i_k}, MINPE(n_{i_k})) + c_{i_k i}\}$$

where n_i has p parent nodes and n_{i_k} is the k-th parent node. The parent node that maximizes the above expression is called the Very-Important-Parent of n_i and is denoted by $VIP(n_i, J)$.

Given the communication-constrained earliest start time of a node on a processor, the following axiom governs the decision of whether the node can be scheduled on that processor.

Axiom I: *A node n_i can be scheduled to a processor J on which the set of nodes $\{n_{J_1}, \ldots, n_{J_m}\}$ has been scheduled iff there exists some k such that*

$$ST(n_{J_{k+1}}, J) - max\{FT(n_{J_k}, J), CEST(n_i, J)\} \geq w(n_i)$$

where $k = 0, \ldots, m$; $ST(n_{J_{m+1}}, J) = \infty$; and $FT(n_{J_0}, J) = 0$.

Intuitively, the axiom implies that a node cannot be schedule to a processor unless that processor has an idle time slot large enough to accommodate the node. In case the node can be scheduled, Axiom II given below determines its actual start time.

Axiom II: *The earliest start time of n_i on processor J, denoted by $EST(n_i, J)$, is $max\{CEST(n_i, J), FT(n_{J_l}, J)\}$ where l is the minimum value of k satisfying the inequality in Axiom I. If there does not exist such l, $EST(n_i, J)$ is defined as ∞.*

It should be noted that both $CEST(n_i, J)$ and $EST(n_i, J)$ are varying quantities; their values depend on the current state of scheduling.

2.2 Related Algorithms

Using duplication in static task scheduling is a relatively unexplored research topic. Kruatrachue and Lewis [5] have proposed one such scheduling algorithm, called *Duplication Scheduling Heuristic (DSH).* Another algorithm, called *Bottom-up-Top-down Duplication Heuristic (BTDH),* has been recently proposed by Chung and Ranka [1].

In our opinion, the DSH algorithm has the following deficiency. As it considers only the idle time slot between the finish time of the last node scheduled to a processor and the earliest start time of the candidate node (the one being considered for scheduling), the degree of duplication is likely to be small. Thus, duplication may not always be effective.

The BTDH algorithm is essentially an extension of the DSH algorithm. The complexity of both algorithms is $O(n^4)$. There are basically two differences between them.

i) The BTDH algorithm does not indicate any preference as to which parent node to be considered for duplication.

ii) The duplication process does not stop as long as the idle time slot has not been overflown. That is, the process does not stop even if the start time of the candidate node is increased.

Despite its better performance, the BTDH algorithm has the following drawback. The algorithm may duplicate some parent nodes which will not reduce the start time of a node. Thus, at later steps, when the algorithm considers the most important parent node — the one from which the data sent arrives last, there may be no space in the idle time slot to accommodate it.

3 The Proposed Algorithm

In this section, we describe our proposed scheduling algorithm. We make two assumptions in our study. First, we assume that the processor network is fully-connected with unlimited number of identical processors. Second, each processor has dedicated hardware to deal with communication so that communication can take place simultaneously with computation. Before describing the algorithm, we discuss some of the basic principles used in its design.

3.1 Design Principles

At each scheduling step, some nodes are more important so they should be given higher priorities which in turn means that they should be scheduled first. Determining node priorities requires an attribute, which is given by the following definition.

Definition 2: *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 cost and communication cost is the maximum.*

Proper scheduling of nodes on the CP can potentially generate efficient schedules. However, we need to schedule the parent nodes of CPNs efficiently also. The following definition explains a partitioning of nodes which can be used to assign accurate priorities to nodes.

Definition 3: *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.*

Clearly, in order not to violate the precedence constraints among nodes, all IBNs of each CPN have to be scheduled before the CPN is considered for scheduling. The OBNs need to be scheduled in an efficent manner also. The following definition gives a way to efficiently schedule the OBNs.

Definition 4: *The OBN Binding is an ordering of OBNs such that an OBN n_i has a higher priority than another OBN n_j if n_i's depth is larger than n_j's, under the constraint that the parent node of an OBN n_i, which is also an OBN, always has higher priority than n_i.*

The duplication technique used in our proposed algorithm is different from other algorithms. We duplicate the ancestor nodes of each CPN, which may be CPNs or IBNs, in descending order of message arrival times. Thus, the more important parent nodes are always duplicated first. In addition, we apply the duplication technique

recursively upward from the parent nodes so that the CPN being considered can potentially start at the earliest possible time. The following rule formalizes the duplication technique.

Duplication Rule (DR):

Suppose that n_i is being considered to schedule on processor J. The duplication node list (DNL) for n_i on processor J as well as the EST(n_i, J) are determined in the following steps.

1) *Determine EST(n_i, J).*
2) *If EST(n_i, J) = ∞ or VIP(n_i, J) does not exist or VIP(n_i, J) is scheduled on J, then the start time of n_i cannot be reduced by duplication. The duplication process stops at this step.*
3) *Otherwise, insert VIP(n_i, J) into DNL(n_i, J) provided EST(n_i, J) does not increase. If this VIP is not inserted, the duplication process terminates; otherwise, replace n_i by VIP(n_i, J) and repeat the process from step 1).*

3.2 The CPFD Algorithm

The proposed algorithm, which applies duplication to schedule CPNs efficiently, is called *Critical Path Fast Duplication* (CPFD) algorithm. The algorithm uses two procedures: *Attempt_Duplication*, and *Trace_Ancestor*. They are described below.

***Attempt_Duplication(n_i)*:**

(1) *min_EST* ← ∞, *min_PE* ← NULL, *min_DNL* ← NULL
(2) Push on *PE_Stack*: (i) an unused processor; (ii) all processors containing the parent nodes of n_i.
(3) **while** *PE_Stack* is not empty **do**
(4) J ← top of *PE_Stack*
(5) Apply the DR to n_i
(6) **if** *EST*(n_i, J) < *min_EST* **then**
(7) *min_EST* ← *EST*(n_i, J), *min_PE* ← J, *min_DNL* ← *DNL*(n_i, J)
(8) **end if**
(9) **end while**
(10) Duplicate nodes on *min_PE* according to *min_DNL*
(11) Schedule n_i to *min_PE* with *ST*(n_i, *min_PE*) ← *min_EST*

Attempt_Duplication works by constructing a stack of candidate processors to find the one which gives the lower bound start time of n_i. The complexity of *Attempt_Duplication* is determined as follows. Step 5 is the dominant step. This step takes $O(en)$ time. There are at most $O(p)$ execution of this step.Thus, the complexity of *Attempt_Duplication* is $O(pen)$.

***Trace_Ancestor(n_i)*:**

(1) **while** there exists unscheduled parent node of n_i **do**
(2) n_p ← an unscheduled parent node of n_i
(3) *Trace_Ancestor*(n_p)
(4) **end while**
(5) *Attempt_Duplication*(n_i)

Trace_Ancestor works by recursively scheduling all the parent nodes (and other ancestor nodes as well) before scheduling n_i itself. The complexity of *Trace_Ancestor* is $O(mpen)$ if there are $O(m)$ unscheduled ancestor nodes. Based on *Attempt_Duplication* and *Trace_Ancestor*, the CPFD algorithm is formalized below.

The CPFD Algorithm:

(1) Determine a CP. Break ties by selecting the one with a larger sum of computation costs.
(2) **for** each CPN n_i (start from the entry node) **do**
(3) *Trace_Ancestor*(n_i)
(4) **end for**
(5) Perform OBN Binding
(6) **for** each OBN n_j (start from the one with the highest priority) **do**
(7) *Trace_Ancestor*(n_j)
(8) **end for**

The complexity of the CPFD algorithm is $O(pen^2)$ as there is $O(n)$ execution of *Trace_Ancestor*. Thus, the CPFD algorithm is practical even for large task graphs.

3.3 An Application Example

In this section, we illustrate the effectiveness of the CPFD algorithm by showing its schedule for a randomly generated task graph. For comparison, the schedules produced by the DSH and BTDH algorithms are also presented.

Both DSH and BTDH algorithms need a supplementary scheduling algorithm to determine the priorities of nodes. It is shown in [1] that the Highest Level First with Estimated Time (HLFET) scheduling algorithm [8] gives better results. The HLFET algorithm, which is an extension of Hu's classic work [3], defines the priority of a node as the largest sum of computation costs among all the directed path from the node to an exit node in the task graph. In what follows, we call the two algorithms as DSH/HLFET and BTDH/HLFET, respectively, to indicate that they employ the HLFET algorithm to determine priorities of nodes.

A schedule for the random task graph (Figure 1 (a)) without duplication is shown in Figure 1(b). The schedule length is 301 time units. This schedule, which is generated by hand, is the best possible schedule without duplication. The lower bound is 246 time units, which is equal to the sum of computation costs along the CP, cannot be achieved in this case. Nodes n_6, n_7, n_8, n_9 already start at the earliest possible time and they cannot start earlier because of the unavoidable communication delays.

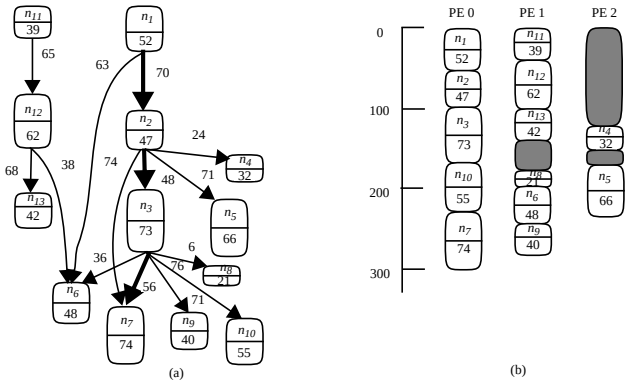

Figure 1: (a) A randomly generated task graph; (b) The best possible schedule without duplication (schedule length = 301).

The schedule generated by DSH/HLFET is shown in Figure 2(a). The communication edges are not shown for clarity. The schedule length is 275 time units. The problem with the DSH algorithm is revealed by the scheduling of n_{10}. If no parent node is duplicated to PE 3, the start time of n_{10} will be at time 248. Thus, n_3 is duplicated to processor PE 3 and the start time of n_{10} reduces to 220. However, n_2 is not duplicated to PE 3 because this would increase the start time of n_3 and hence that of n_{10} to time 169 and time 242 respectively. Thus, according to the DSH algorithm, n_2 is not duplicated and the duplication process terminates at that point. On the other hand, if n_1 is also duplicated to PE 3, the start time of n_2, n_3 and n_{10} will be reduced dramatically. This can be seen from the schedule generated by BTDH/HLFET shown in Figure 2(b). However, the schedule produced by BTDH/HLFET is still

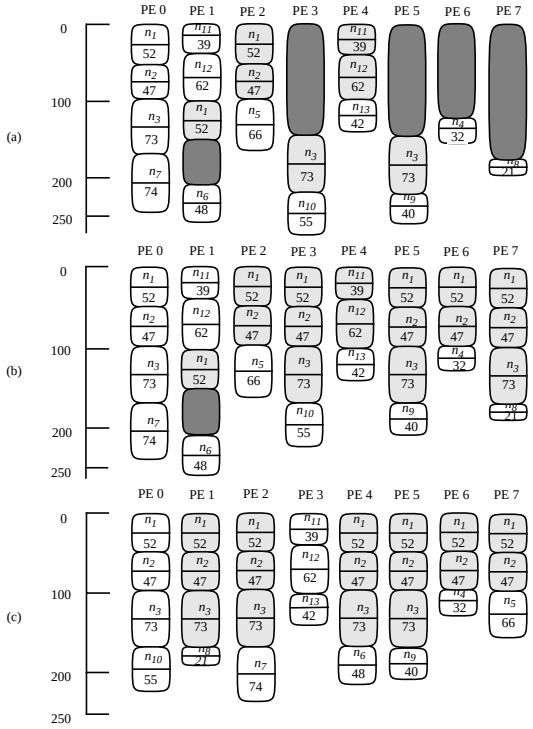

Figure 2: Schedules of the random task graph generated by (a) DSH/ HLFET (schedule length = 275); (b) BTDH/HLFET (schedule length = 258); (c) CPFD (schedule length = 246).

not the best. Consider the nodes assigned to PE 1. Although the parent nodes n_1 and n_{12} are duplicated to PE 1, the start time of n_6 is still not improved because it has to wait for the data from n_3. The node n_3 is not duplicated since the time slot on PE 1 is not large enough to accommodate it. The schedule produced by CPFD is shown in Figure 2(c). The schedule length is 246 time units which is the best possible. All nodes are able to start at the earliest possible times due to proper duplication. The problems with DSH and BTDH do not occur with the CPFD algorithm.

4 Performance and Comparison

To test and compare the performance of the proposed scheduling algorithm, we generated a suite of task graphs. Our objective is to compare the schedule lengths produced by all three algorithms for various graph structures, different values of CCR and the task graph size in terms of the number of nodes.

4.1 Workload

We generated task graphs with seven different types of structures: completely random graphs, in-tree graphs, out-tree graphs, fork-join graphs and task graphs correspond to three parallel algorithms — Gaussian elimination, LU-decomposition and Laplace Equation Solver. Within each type of graph structure, we chose seven values of CCR which are 0.1, 0.5, 1.0, 1.5, 2.0, 5.0 and 10.0. For each of the seven values of CCR, we generated 10 different graphs with the number of nodes varying from 10 to 100 with an increment of 10. This implies that for each value of CCR, there are 70 graphs, and the total number of graphs is 490. For each graph, the weights of the nodes and the communication edges are different and have been chosen randomly such that the average CCR of the graph corresponds to one of the seven values of CCR described above.

Table I: A performance comparison of the three scheduling algorithms.

			The worst % degradation in schedule length	The maximum% improvement in schedule length	The average% improvement in schedule length	Number of times it performs the same	Number of times it performs worse	Number of times it performs better
BTDH compared with DSH	CCR	0.1	3	4	63	-0.10	0.37	0.76
		0.5	12	0	58	0.57	2.48	None
		1.0	16	2	52	1.62	4.19	0.33
		1.5	15	0	55	2.17	6.89	None
		2.0	22	1	47	2.79	7.86	1.39
		5.0	26	0	44	6.19	17.74	None
		10.0	30	0	40	8.92	19.07	None
CPFD compared with DSH	CCR	0.1	27	0	43	1.56	5.48	None
		0.5	39	0	31	2.66	6.40	None
		1.0	56	0	14	4.72	8.93	None
		1.5	54	0	16	6.07	12.83	None
		2.0	52	0	18	7.35	13.63	None
		5.0	42	0	28	7.49	17.99	None
		10.0	45	0	25	10.47	20.54	None
CPFD compared with BTDH	CCR	0.1	27	0	43	1.66	5.36	None
		0.5	34	0	36	2.11	4.85	None
		1.0	43	0	27	3.11	5.87	None
		1.5	43	0	27	3.92	7.51	None
		2.0	42	0	28	4.56	8.63	None
		5.0	28	0	42	1.33	3.82	None
		10.0	28	0	42	1.91	5.31	None

4.2 Relative Performance

Table I summarizes the relative performance of the DSH, BTDH and CPFD algorithms in terms of the schedule lengths produced for the suite of task graphs. There are three types of comparisons given in this table. First, we use DSH as the reference and compare the performance of BTDH and CPFD relative to it. Next, we compare the performance of CPFD with BTDH. For each comparison, there are seven rows in the table, with each row corresponding to results of running the scheduling algorithms on 70 different task graphs for that value of CCR. The first three columns indicate the comparative performance of the scheduling algorithms in terms of schedule lengths of these 70 graphs. For example, when BTDH is compared with DSH when CCR is equal to 0.1, the first row in the table indicates that BTDH generated a shorter schedule length on 3 graphs, generated a longer schedule on 4 graphs while the schedule length on 63 graphs was the same for both algorithms. Similarly, the next three columns indicate the average percentage improvement, maximum percentage improvement and average percentage degradation in the schedule length produced by BTDH over DSH. These numbers have been taken across the schedule lengths of 70 graphs for each value of CCR. An inspection of Table I reveals that BTDH performs increasingly better than DSH for higher values of CCR. Also, BTDH yields better value of the average percentage improvement and maximum percentage improvements. However, there are occasional cases when BTDH performs worse than DSH. When CPFD is

compared against DSH, it is immediately apparent that the number of times it performs better is increased not only for the larger values of CCR but also for smaller values of CCR. The average improvement in the schedule varies from 1.56 to 10.47%. The CPFD algorithm also outperforms BTDH for all values of CCR. The average percentage improvement in the schedule length varies from 1.33 to 4.56%. There is no single case out of 490 tests, where CPFD performs worse than DSH or BTDH.

4.3 Absolute Performance

The results providing the relative performance described above are supplemented by the results showing the performance of each algorithm with respect to the lower bound on the schedule length. This bound, which is the sum of the computation costs of the nodes on the CP, provides a lower limit on the schedule length. The lower bound, however, may not be achievable with any scheduling algorithm and the optimal schedule length may well exceed this bound. When scheduling the test graphs, we observed the number of times each algorithm produced a schedule length equal to the lower bound. The bar charts shown in Figure 4 indicate the number of times lower

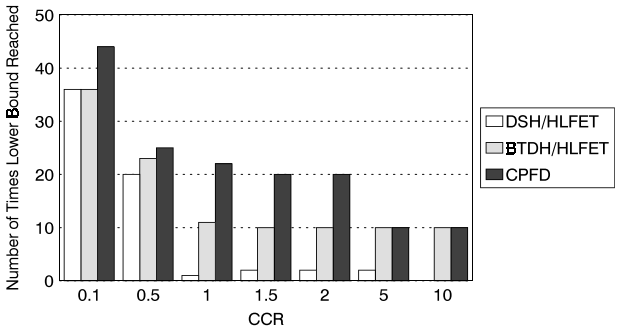

Figure 3: The Number of times lower bound on the schedule length achieved with each algorithm.

bound was achieved with each algorithm for different values of CCR. Again, there are 70 test cases for each CCR value. As expected, lower bound is more likely to be achieved when the value of CCR is low. One noticeable point is that DSH rarely reaches the lower bound if the value of CCR is 1.0 or higher. In contrast, BTDH is still able to achieve lower bound in 10 out of 70 test cases. The CPFD algorithm, on the other hand, performs much better than both DSH and BTDH at lower as well as higher values of CCR. The CPFD algorithm achieved lower bounds on all out-tree graphs. Figure 4 shows the average normalized schedule lengths produced by each algorithm with the number of nodes in each graph varying from 10 to 100. The normalized schedule length, which is defined as the actual schedule length divided by the lower bound, increases a little bit with the graph size. This is because the proportion of nodes which are not on the critical path slightly increases as the graph size increases. Thus, the lower bound, which is determined by the CP, becomes less likely to reach. The performance of the CPFD algorithms is consistently superior than the other two algorithms for different graph sizes. Furthermore, the difference between the normalized schedule length of CPFD and the other two algorithms tends to increase for larger graphs.

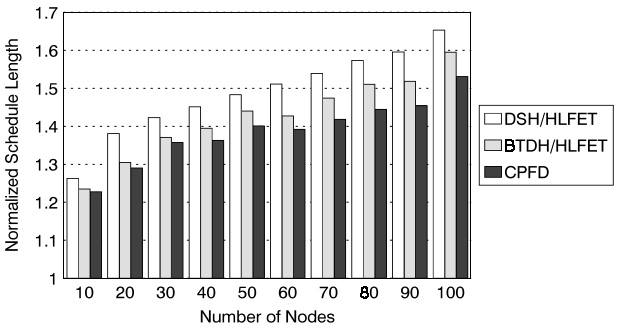

Figure 4: Normalized schedule lengths for each algorithm with respect to the lower bound.

5 Conclusions

Using task duplication in scheduling can be useful especially when the CCR of a parallel algorithm on a given system is high. This is usually the case in distributed systems such as cluster of workstations. Both DSH and BTDH algorithms produce good solutions with the latter outperforming the former when CCR is very high. However, the basic principle in both the algorithms is essentially the same, that is, to duplicate a parent task if it improves start time of a node. The proposed CPFD algorithm which uses a new technique tries to start every tasks at the earliest possible time from the beginning of the scheduling process. The proposed algorithm outperforms both of these algorithms without performing worse in any of the 490 test cases. Moreover, it consistently performs better at low as well as high values of CCR.

References

[1] 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. of *Supercomputing'92*, Nov. 1992, pp.512-521.

[2] M.R. Gary and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979, W.H. Freeman and Company.

[3] T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Oper. Research*, vol. 19, no. 6, Nov. 1961, pp.841-848.

[4] S.J. Kim and C. Browne, "A General Approach to Mapping of Parallel Computations upon Multiprocessor Architectures," Proc. of *Int'l Conf. on Parallel Processing*, vol. 3, Aug. 1988, pp.1-8.

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

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

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

[8] C.V. Ramamoorthy, K.M. Chandy and M.J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. on Computers*, vol. C-21, Feb. 1972, pp.137-146.

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

[10] G. Sih and E. Lee, "Declustering: A New Multiprocessor Scheduling Technique," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 6, Jun. 1993, pp.625-637.