A Comparison of Task-Duplication-Based Algorithms for Scheduling Parallel Programs to Message-Passing Systems†

Ishfaq Ahmad and Yu-Kwong Kwok

Department of Computer Science
The Hong Kong University of Science and Technology, Hong Kong
Email: {iahmad, csricky}@cs.ust.hk

Abstract

A major hurdle in achieving high performance in message-passing architectures is the inevitable communication overhead that occurs when tasks scheduled on different processors need to exchange data. This overhead can cause a stern penalty especially in distributed systems such as clusters of workstations, where the network channels are considerably slower than the processors. For a given parallel program represented by a task graph, the communication overhead can be mitigated by redundantly executing some tasks on which other tasks critically depend. There have been a few task-duplication based scheduling algorithms that are designed for such environments. Although these algorithms are independently shown to be effective, no attempt has been made to quantitatively compare their performance under a broad range of input parameters. In this paper we analyze the problem of using task-duplication in compile-time scheduling of task graphs on parallel and distributed systems. We discuss the characteristics of six recently proposed algorithms, and examine their merits, differences, and expediency for different environments. Through a comprehensive experimental evaluation, the six algorithms are compared in terms of schedule lengths, number of processors used, and the amount of scheduling time required.

Keywords: Algorithms, Distributed Systems, Multicomputers, Multiprocessors, Duplication-Based Scheduling, Parallel Scheduling, Task Graphs.

1 Introduction

Recently we have witnessed great advancements in multicomputer architectures but interprocessor communication overhead is nevertheless a major hurdle to efficient execution of parallel programs. This overhead occurs when two tasks of a parallel program are assigned to different processors but need to exchange some data. However, such adverse effects can be alleviated by using task-duplication based scheduling techniques. Task-duplication means scheduling a parallel program by redundantly allocating some of its tasks on which other tasks of the program critically depend. This can potentially reduce the start times of tasks waiting to receive the data from the critical tasks, and eventually improves the overall execution time of the entire program.

In this paper we address the static scheduling problem with task-duplication. In static scheduling, a parallel program can be represented by a directed acyclic graph (DAG) with 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 a DAG represents a task which is a set of instructions that must be executed sequentially in the same processor without preemption. Associated with each node is a number indicating the computation time required. This number is called

†. This research was supported by a grant from the Hong Kong Research Grants Council under contract number HKUST 734/96E.

the *computation cost* of a node n_i and is denoted by $w(n_i)$. The directed edges in the DAG correspond to the communication messages and precedence constraints among the tasks. Associated with each edge is a number indicating the time required to communicate the data. This number is called the *communication cost* of the edge and is denoted by $c(n_i, n_j)$. The *communication-to-computation-ratio* (*CCR*) of a DAG is defined as its average communication cost divided by its average computation cost. The target multicomputer architecture is modeled as a fully-connected network of *processing elements* (PEs). Each PE consists of a processor and a local memory so that the PEs share data solely by message-passing. As such, the communication cost among two nodes assigned to the same PE is assumed to be zero. If a node n_i is scheduled to some PE, $ST(n_i)$ and $FT(n_i)$ denote its start time and finish time, respectively. After all nodes have been scheduled, the *schedule length* is defined as $\max_i\{FT(n_i)\}$ across all processors. The objective of a scheduling algorithm is to minimize $\max_i\{FT(n_i)\}$ by properly allocating the nodes to the PEs and sequencing their execution orders without violating the precedence constraints.

Scheduling a DAG to multiprocessors is an NP-complete problem in most cases [5], [7], [16] so that many heuristics have been suggested [1], [12], [14], [18], [19]. The complexity and solution quality of a heuristic largely depend on the structure of the DAG and the target machine model [1], [5], [8], [9], [10], [11], [17].

Even with an efficient scheduling algorithm, it may happen that some PEs are idle during different time slots because some nodes wait for the data from the nodes assigned to other PEs. If these idle time slots can be utilized effectively by identifying and redundantly allocating the critical nodes, the schedule length can be further reduced. For example, consider a simple task graph shown in Figure 1(a). An optimal schedule without duplication is shown in Figure 1(b) (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 Figure 1(c). Thus, duplication has the potential to reduce the schedule length by efficiently utilizing the processors. Using duplication, however, also makes the scheduling problem more complex since the scheduling algorithm, in addition to observing the precedence constraints, needs to select important nodes for duplication and identify idle time slots to accommodate those nodes.

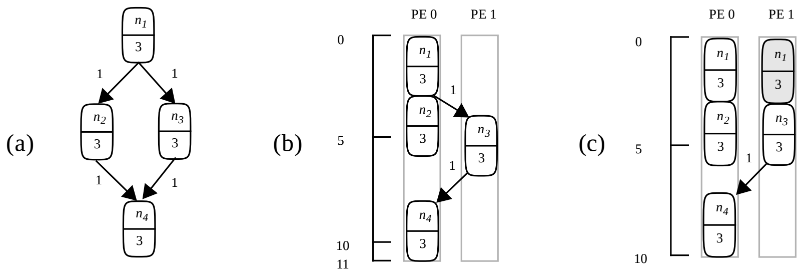


Figure 1: (a) A simple task graph; (b) a schedule without duplication; (c) a schedule with duplication.

In this paper we present an extensive comparison of six recently reported task-duplication based algorithms including: the DSH (Duplication Scheduling Heuristic) algorithm [13], the BTDH (Bottom-up Top-down Duplication Heuristic) [4], the LWB (Lower-Bound) algorithm [6], the PY algorithm (named after its designer Papadimitriou and Yannakakis) [16], the LCTD (Linear-Clustering with Task-Duplication) algorithm [18], and the CPFD (Critical-Path Fast Duplication) algorithm [2]. We use four suites of DAGs with a broad range of CCRs and graph sizes. The algorithms are evaluated in terms of schedule lengths, number of processors used, and scheduling time required. The characteristics of the six algorithms compared in this paper are briefly shown in Table 1.

This paper is organized as follows. In Section 2, we present an example to illustrate the functionality of all algorithms. In Section 3, we present the performance results of all the algorithms. Some concluding remarks are provided in the last section.

Table 1: Some duplication-based scheduling algorithms and their characteristics.

Algorithm	Researchers	Ancestors Duplicated	Optimality	Complexity
DSH	Kruatrachue *et al.* [13]	All possible ancestors	Unknown	$O(v^4)$
PY	Papadimitriou *et al.* [16]	All possible ancestors	Within a factor of 2 from optimal	$O(v^2(e + v\log v))$
LWB	Colin *et al.* [6]	Only ancestors on the same path	Optimal for DAGs with computation costs strictly greater than communication costs	$O(v^2)$
BTDH	Chung *et al.* [4]	All possible ancestors	Unknown	$O(v^4)$
LCTD	Chen *et al* [18]	All possible ancestors	Unknown	$O(v^3\log v)$
CPFD	Ahmad *et al.* [2]	All possible ancestors	Optimal for in-trees and DAGs with computation costs greater than comm. costs	$O(ev^2)$

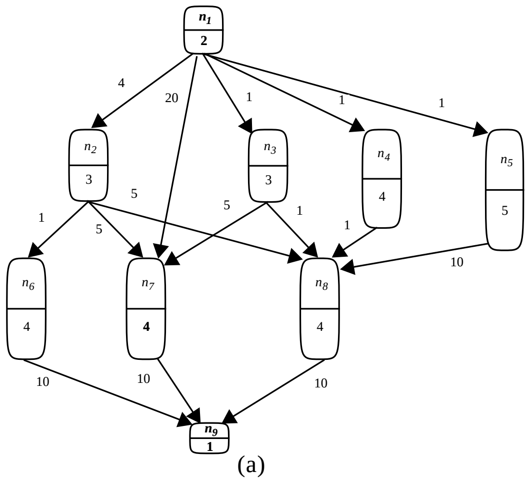
2 Task-Duplication-Based Scheduling

In order to have a qualitative understanding of the functionality of the six algorithms, in this section we present a scheduling example by using a small DAG (see Figure 7(a)). Some node attributes used by the algorithms for computing priorities are also shown in Figure 7(b). Note that the nodes marked by an asterisk are called *critical path* nodes (CPNs). A critical path (CP) is a path in the DAG with the largest sum of computation and communication costs. The CPNs are commonly reckoned as more important nodes because timely scheduling of the CPNs can potentially lead to a shorter schedule. The SL of a node is the largest sum of computation costs on a path from the node to an exit node. Further, the list used by the CPFD algorithm, called the CPN-Dominant Sequence (denoted by CPN-DS), is also shown in Figure 7(b). For details of the construction of CPN-DS, the reader is referred to [2].

A schedule without task duplication is shown in Figure 7(c). 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 3 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 overall schedule length can be reduced.

The schedule generated by the LWB algorithm is shown in Figure 3 which also includes a scheduling trace. The schedule length is reduced by 1 time-unit 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. The problem is that the LWB 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 dramatically reduced if the nodes n_2 and n_3 are also duplicated to PE 4. The main drawback of the LWB algorithm is that it only considers the ancestors on the same path for scheduling a node. When a node has more than one heavily communicated parents, this technique does not minimize the start time of the node (which can be done by duplicating more than one parents on a processor).

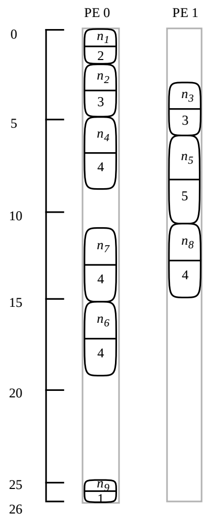
The schedule generated by the LCTD algorithm and the scheduling trace are shown in Figure 4. The schedule length is much shorter than that of the LWB algorithm and the utilization of processors is also much better. This is because the LCTD algorithm considers every ancestor nodes reaching a node for



(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

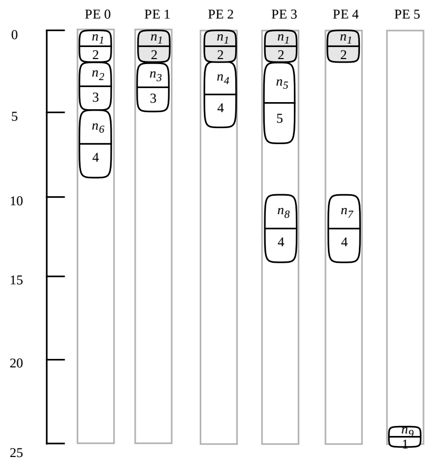
(b)



(c)

Figure 2: (a) A simple task graph; (b) static levels of the nodes; (c) a schedule without using task duplication (schedule length = 26).

(a)

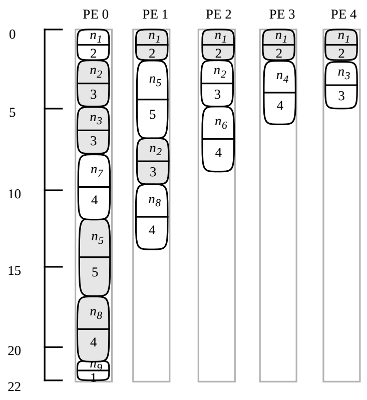


(b)

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

Figure 3: (a) The schedule generated by the LWB algorithm (schedule length = 25); (b) The lower bound values computed by the LWB algorithm.

(a)



(b)

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

Figure 4: (a) The schedule generated by the LCTD algorithm (schedule length = 22); (b) A scheduling trace of the LCTD algorithm.

duplication.

The DSH and BTDH algorithms generate the same schedule shown in Figure 5. This is presumably because the BTDH algorithm is essentially an extension of the DSH algorithm. Although the schedule length is the same as that of the LCTD algorithm, the scheduling of most of the 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 algorithm examines the nodes for scheduling in a descending order of static levels. Linear clustering can easily make mistake in identifying 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 mistake can be detrimental because the start times of some critical nodes may well be reduced by using a new processor in which its ancestors are duplicated. Thus, this lack of space for duplicating ancestors of a node causes duplication ineffective.

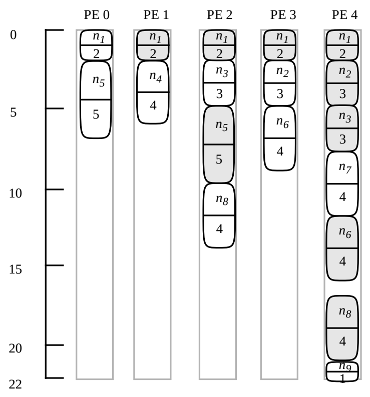
The schedule generated by the PY algorithm is shown in Figure 6. The schedule length is much longer than that of the previous three algorithms, despite that the PY algorithm guarantees the schedule length to be within a factor of 2 from the optimal. It should be noted that the schedule length is even longer than the schedule without duplication shown earlier. The principles used by the PY algorithm are again entirely different in that it relies on the lower bound values (the *e* values) to select the nodes from the sub-graph for duplication. The *e* value of a node is computed by iteratively constructing the largest sub-graph reaching the node. The nodes included in the sub-graph are candidates for duplication. The major problem is that it clusters nodes in a sub-graph for duplication by merely 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 that are more important to reducing the start time of the given node and lead to a poor schedule.

The schedule generated by the CPFD algorithm is shown in Figure 7(a). The schedule length is 20 which is optimal. The scheduling steps are also shown in Figure 7(b). Notice that the node(s) duplicated at each step is also shown in the last column of the table. At the first step, the first CPN, 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, their start times are recursively minimized. Next, the recursive 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 so that they are scheduled to start at the earliest possible times. Finally, when n_9 is scheduled, only the necessary nodes 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.

3 Performance and Comparisons

The six algorithms were implemented on a SUN SPARC Station 2 using the C language. We used two suites of task graphs: regular and irregular task graphs. The regular graphs represent two parallel applications: the mean value analysis [3] and the LU-decomposition [14]. The irregular graphs include the in-tree, out-tree, fork-join and completely random task graphs [3].

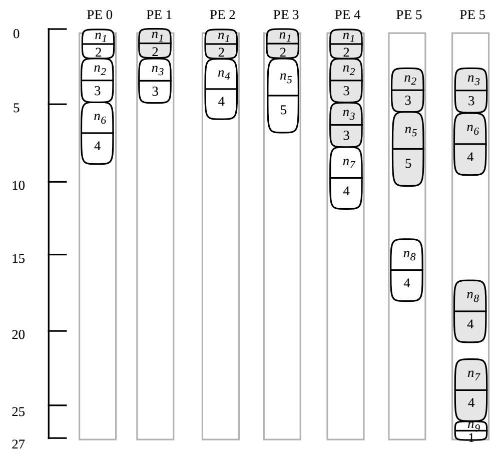
In each graph, the computation costs of the individual nodes were randomly selected from a uniform distribution with mean equal to the chosen average computation cost. Similarly, the communication costs of the edges were randomly selected from a uniform distribution with mean equal to the average communication cost. Within each type of graph, we used seven values of CCR which were 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 an increment of 50. The regular graphs on the other hand can be characterized by the size of their data matrix (the number of nodes is roughly

(a) 

(b)

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

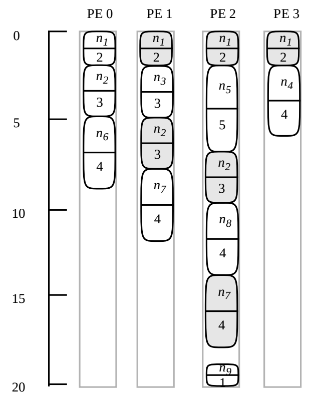
Figure 5: (a) The schedule generated by the DSH and BTDH algorithms (schedule length = 22); (b) A scheduling trace of the DSH and BTDH algorithms.

(a) 

(b)

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

Figure 6: (a) The schedule generated by the PY algorithm (schedule length = 27); (b) The *b* values computed by the PY algorithms.

(a) 

(b)

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

Figure 7: (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; an entry with “N.C.” indicates the processor is not considered).

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 420 (6 graph types, 7 CCRs, 10 graph sizes).

The performance comparison of the CPFD, LWB, LCTD, DSH, BTDH and PY algorithm was made 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. Next, we compared the number of times each algorithm produced the best solution. We also observed the percentage degradation in performance of an algorithm compared to the best solution. Finally, the running times by each algorithm were also noted.

For the first comparison, we present the schedule lengths produced by the six algorithms. The normalized schedule length (NSL), defined as the ratio of schedule length to the critical-path length, is computed for each solution generated by the algorithms.

In Table 2(a), the average NSLs produced by each algorithm for each graph type (averaged over 7 values of CCR and 10 different graph sizes) are shown. These numbers clearly indicate that the CPFD algorithm produced the shortest average schedule length not only across all graphs types but also for

	Graph Types					
Algorithm	LU	MVA	InTree	OutTree	ForkJoin	Random
LWB	1.32	2.22	2.49	1.00	2.20	2.12
LCTD	1.34	2.01	2.10	1.48	1.97	1.86
DSH	1.21	1.60	1.83	1.22	1.65	1.65
BTDH	1.21	1.55	1.79	1.00	1.58	1.56
PY	1.49	1.93	2.11	1.24	2.29	2.06
CPFD	1.15	1.52	1.76	1.00	1.54	1.50

Table 2(a): Average NSLs across all graph types.

	CCRs						
Algorithm	0.1	0.5	1.0	1.5	2.0	5.0	10.0
LWB	1.01	1.08	1.24	1.40	1.57	2.64	4.52
LCTD	1.01	1.13	1.33	1.56	1.76	2.50	3.33
DSH	1.02	1.10	1.23	1.33	1.43	1.89	2.47
BTDH	1.03	1.10	1.21	1.30	1.40	1.77	2.26
PY	1.03	1.18	1.37	1.55	1.73	2.58	3.39
CPFD	1.01	1.07	1.17	1.26	1.34	1.75	2.21

Table 2(b): Average NSLs across all CCRs.

	Matrix Dimensions									
Algorithm	15	16	17	18	19	20	21	22	23	24
LWB	1.55	1.62	1.68	1.73	1.78	1.81	1.85	1.89	1.92	1.95
LCTD	1.35	1.45	1.52	1.58	1.65	1.70	1.75	1.79	1.83	1.87
DSH	1.23	1.27	1.31	1.33	1.35	1.37	1.40	1.41	1.43	1.44
BTDH	1.23	1.26	1.30	1.32	1.34	1.36	1.38	1.40	1.42	1.43
PY	1.43	1.56	1.57	1.60	1.63	1.68	1.69	1.73	1.74	1.75
CPFD	1.21	1.24	1.26	1.28	1.30	1.31	1.32	1.33	1.35	1.36

Table 2(c):Average NSLs for regular task graphs of various matrix dimensions.

	Number of Nodes									
Algorithm	50	100	150	200	250	300	350	400	450	500
LWB	1.62	1.76	1.76	1.85	1.93	1.98	2.06	2.07	2.05	2.16
LCTD	1.39	1.60	1.61	1.75	1.82	1.82	1.95	2.01	2.01	2.08
DSH	1.27	1.42	1.38	1.46	1.49	1.52	1.55	1.61	1.60	1.67
BTDH	1.24	1.37	1.32	1.40	1.45	1.44	1.49	1.53	1.53	1.61
PY	1.54	1.69	1.70	1.77	1.82	1.89	1.96	1.97	1.96	2.03
CPFD	1.23	1.35	1.30	1.37	1.41	1.41	1.44	1.47	1.47	1.55

Table 2(d): Average NSLs for irregular task graphs of various sizes.

each type of graph. The next best algorithm was the BTDH algorithm. There was a little difference between the performance of DSH and BTDH. The LWB algorithm had a large variations in its performance. It performed good for LU-decomposition DAGs and out-trees but did not perform well on other graphs. Based on this comparison, the ranking of the six algorithms is as follows: CPFD, BTDH, DSH, LCTD, PY, and LWB.

Table 2(b) shows the NSLs (averaged over graph size and graph type) of each algorithm against various values of CCR. We can observe that all algorithms were very sensitive to the value of CCR. This is because a larger value of CCR implies a larger variation in the start times of the nodes, which in turn causes the algorithms to make more wrong scheduling decisions. A large value of CCR can thus test the robustness of an algorithm. We can also notice that the differences between the performance of various algorithms became more significant with larger values of CCR. The ranking of the algorithms based on performance, however, is consistent with our earlier conclusion.

Table 2(c) and Table 2(d) show the NSLs yielded by each algorithm against various graph types (averaged across graph types and CCRs), for regular and irregular graphs, respectively. We can notice that in this context the CPFD was also consistently better than all other algorithms. The size of the graph, both for regular and irregular graphs, had no bearing on this conclusion.

Table 3(a) shows the number of times an algorithm produced the best solution compared to all other algorithms (each entry is the number of best solutions out of 70 trials with various values of CCR and graph sizes) for various types of graphs. The right most column shows the total number of best solutions out of 420trials. The CPFD algorithm produced the best solutions 413 times out of 420 cases. The LWB algorithm produced the next highest number (211 cases) of best solutions. The BTDH algorithm showed comparable performance as that of the LWB algorithm. The other algorithms, however, generated dramatically smaller number of best solutions.

Table 3(b) shows the effects of CCR on an algorithm's ability to produce the best solution. The LBW and LCTD algorithms can be seen to perform well only with low values of CCR. The rest of the algorithms appeared to be insensitive to the value of CCR. The impact of the graphs size, as shown in Table 3(c) and Table 3(d), did not have any effect on an algorithms's ability to produce the best solution.

The next four sets of tables (Table 4 to Table 7) indicate how *badly* an algorithm performed when it could not produce the best solution. That is, for each test case, we compared the schedule length produced by an algorithm with the best solution yielded by another algorithm, and measured the amount of degradation. Table 4(a) and Table 4(b) show the number of cases when the percentage degradation was less than or equal to 5%. Table 5(a) and Table 5(b) show the number of cases when this degradation was between 5 to 10%. Table 6(a) and Table 6(b) show the number of cases when the degradation was between 10 to 20%. Finally Table 7(a) and Table 7(b) show the number of cases when the degradation was more than 20%. These tables do not include the results showing the impact of graphs size on these numbers since such an impact was found to be insignificant. These tables indicate that, out of 7 cases in which the CPFD algorithm did not produce the best solution, its performance degradation was less than 5% in 5 cases and was more than 20% in one case only. In contrast, the performance degradation of the other algorithms was found to be in all ranges. The performance degradation of the LCTD and PY algorithms in the range of 20% or more was more frequent compared to the rest of the algorithms. The performance of the BTDH and DSH algorithm was better than the LWB, LCTD and PY algorithms. The LWB algorithm, as noticed earlier, had large fluctuations in performance.

An algorithm's time complexity is an important performance measure. We include here the measured running times of all algorithms running on a SUN SPARC workstation. These times are

Algorithm	LU	MVA	InTree	OutTree	ForkJoin	Random	ALL
	Graph Types						
LWB	50	23	26	70	21	21	211
LCTD	20	0	29	20	2	11	82
DSH	33	10	39	23	7	18	130
BTDH	33	15	42	70	19	25	204
PY	0	0	8	14	1	2	26
CPFD	70	70	68	70	70	65	413

Table 3(a): Number of best solutions across all graph types.

Algorithm	0.1	0.5	1.0	1.5	2.0	5.0	10.0
	CCRs						
LWB	70	59	32	2	24	22	2
LCTD	37	23	8	4	7	2	1
DSH	37	25	9	17	11	11	20
BTDH	36	29	21	38	21	21	38
PY	2	2	3	9	4	3	3
CPFD	60	60	60	57	60	59	57

Table 3(b): Number of best solutions across all CCRs.

Algorithm	15	16	17	18	19	20	21	22	23	24
	Matrix Dimensions									
LWB	8	8	8	7	7	7	7	7	7	7
LCTD	3	1	2	2	2	2	2	2	2	2
DSH	15	13	4	2	2	2	3	1	0	1
BTDH	17	11	5	3	3	3	4	1	0	1
PY	0	0	0	0	0	0	0	0	0	0
CPFD	14	14	14	14	14	14	14	14	14	14

Table 3(c): Number of best solutions for regular task graphs of various matrix dimensions.

Algorithm	50	100	150	200	250	300	350	400	450	500
	Number of Nodes									
LWB	16	14	16	14	13	13	12	14	13	13
LCTD	15	10	8	4	5	3	5	4	5	3
DSH	22	11	13	8	6	7	6	5	6	3
BTDH	28	21	20	17	13	13	11	12	11	10
PY	6	9	3	2	1	0	2	0	2	0
CPFD	28	27	27	26	28	27	27	28	27	28

Table 3(d): Number of best solutions for irregular task graphs of various sizes.

Algorithm	LU	MVA	InTree	OutTree	ForkJoin	Random	ALL
	Graph Types						
LWB	0	7	8	0	11	7	33
LCTD	10	10	8	5	15	11	59
DSH	14	33	16	6	32	17	118
BTDH	14	46	19	0	42	21	142
PY	10	15	19	11	13	11	79
CPFD	0	0	2	0	0	3	5

Table 4(a): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval [0, 5] across all graph types.

Algorithm	0.1	0.5	1.0	1.5	2.0	5.0	10.0
	CCRs						
LWB	2	4	17	9	1	0	0
LCTD	32	11	12	2	1	0	1
DSH	20	26	27	21	8	9	7
BTDH	14	26	25	20	20	19	18
PY	52	17	3	2	1	1	3
CPFD	0	0	0	2	0	0	3

Table 4(b): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval [0, 5] across all CCRs.

	Graph Types						
Algorithm	LU	MVA	InTree	OutTree	ForkJoin	Random	ALL
LWB	0	5	6	0	7	8	26
LCTD	1	8	8	6	5	6	34
DSH	5	23	9	8	15	16	76
BTDH	5	9	7	0	7	20	48
PY	3	9	7	6	10	6	41
CPFD	0	0	0	0	0	1	1

Table 5(a): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval (5, 10] across all graph types.

	CCRs						
Algorithm	0.1	0.5	1.0	1.5	2.0	5.0	10.0
LWB	0	3	5	5	7	6	0
LCTD	0	17	6	5	2	3	1
DSH	3	4	12	12	18	16	11
BTDH	11	7	14	16	11	7	6
PY	6	19	11	3	2	0	0
CPFD	0	0	0	0	0	1	0

Table 5(b): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval (5, 10] across all CCRs.

	Graph Types						
Algorithm	LU	MVA	InTree	OutTree	ForkJoin	Random	ALL
LWB	10	11	5	0	9	10	45
LCTD	15	12	5	10	7	12	61
DSH	14	4	5	11	12	12	58
BTDH	14	0	2	0	2	4	22
PY	9	9	16	13	7	14	68
CPFD	0	0	0	0	0	0	0

Table 6(a): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval (10, 20] across all graph types.

	CCRs						
Algorithm	0.1	0.5	1.0	1.5	2.0	5.0	10.0
LWB	0	6	0	13	14	12	0
LCTD	0	7	21	16	7	4	6
DSH	5	5	6	10	10	9	13
BTDH	3	2	1	6	5	2	3
PY	0	16	20	8	15	4	5
CPFD	0	0	0	0	0	0	0

Table 6(b): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval (10, 20] across all CCRs.

plotted in Figure 8(a) and Figure 8(b), for regular and irregular graphs, respectively. The complexities of these algorithms have been mentioned earlier and concur with the measured timings. The LWB and PY algorithms were found to be faster than the rest of the algorithms. The timings of the CPFD algorithm were slightly slower 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 were also close with the former being slightly slower than the latter.

4 Conclusions

In this paper we have discussed the problem of using task-duplication in scheduling parallel programs represented by DAGs and compared six recently reported algorithms. Through a simple example, the functionality of the six algorithms is illustrated. We have also presented an extensive

	Graph Types						
Algorithm	LU	MVA	InTree	OutTree	ForkJoin	Random	ALL
LWB	10	24	25	0	22	24	105
LCTD	24	40	20	29	41	30	184
DSH	4	0	1	22	4	7	38
BTDH	4	0	0	0	0	0	4
PY	48	37	20	26	39	37	207
CPFD	0	0	0	0	0	1	1

Table 7(a): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval (20, ∞) across all graph types.

	CCRs						
Algorithm	0.1	0.5	1.0	1.5	2.0	5.0	10.0
LWB	0	0	5	6	14	40	40
LCTD	0	0	12	34	43	47	48
DSH	1	0	1	3	8	9	16
BTDH	1	0	0	0	3	0	0
PY	0	9	15	34	45	57	47
CPFD	0	0	0	0	0	0	1

Table 7(b): Number of times the % degradation of NSL (with respect to the best solutions) is within the interval (20, ∞) across all CCRs.

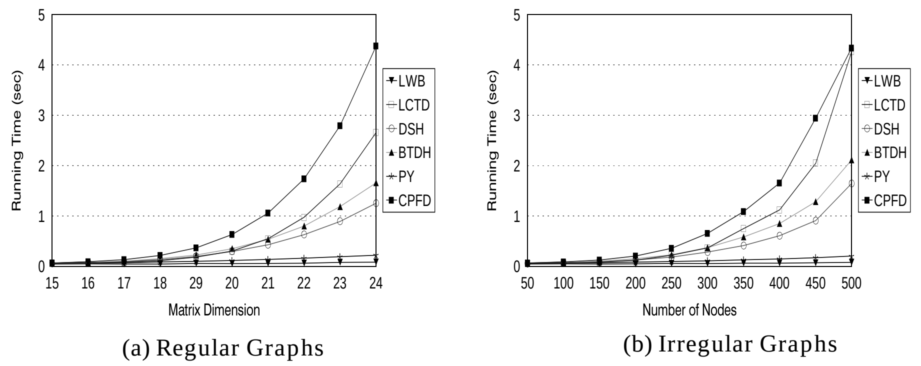


Figure 8: The running time of the six scheduling algorithms on a SPARC Station 2 for (a) regular task graphs of various matrix dimensions and (b) irregular task graphs of various sizes.

comparison of all the six algorithms by noting their performance results under a wide range of input parameters. The evaluation criteria include schedule lengths, number of processors used, and scheduling time required. From the experimental results, it is found that the CPFD algorithm outperformed the other algorithms by a large margin in terms of schedule lengths while it used moderate number of processors and its running times were reasonable. The BTDH and LWB algorithms also showed competitive performance.

References

[1] T.L. Adam, K.M. Chandy, and J. Dickson, “A Comparison of List Scheduling for Parallel Processing Systems,” *Communications of the ACM*, vol. 17, Dec. 1974, pp. 685-690.

[2] I. Ahmad and Y.-K. Kwok, “A New Approach to Scheduling Parallel Programs Using Task Duplication,” Proc. 23rd *Int’l Conf. Parallel Processing*, Aug. 1994, vol. II, pp. 47-51.

[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. of Supercomputing‘92*, 1992, pp. 683-691.

[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. of *Supercomputing’92*, Nov. 1992, pp. 512-521.

[5] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, Wiley, New York, 1976.

[6] J.Y. Colin and P. Chretienne, “C.P.M. Scheduling with Small Computation Delays and Task Duplication,” *Operations Research*, 1991, pp. 680-684.

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

[8] A. Gerasoulis and T. Yang, “On the Granularity and Clustering of Directed Acyclic Task Graphs,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 6, Jun. 1993, pp. 686-701.

[9] M.J. Gonzalez, “Deterministic Processor Scheduling,” *ACM Computing Surveys*, vol. 9, no. 3, Sep. 1977.

[10] 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, 1979, pp. 287-326.

[11] T.C. Hu, “Parallel Sequencing and Assembly Line Problems,” *Operations Research*, vol. 19, no. 6, Nov. 1961, pp. 841-848.

[12] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, “Scheduling Precedence Graphs in Systems with Interprocessor Communication Times,” *SIAM Journal of Computing*, vol. 18, no. 2, Apr. 1989, pp. 244-257.

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

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

[15] R.E. Lord, J.S. Kowalik, and S.P. Kumar, “Solving Linear Algebraic Equations on an MIMD Computer,” *Journal of the ACM*, vol. 30, no. 1, Jan. 1983, pp. 103-117.

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

[17] H. El-Rewini and T.G. Lewis, “Scheduling Parallel Programs onto Arbitrary Target Machines,” *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, Jun. 1990, pp. 138-153.

[18] 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(5), Aug. 1995, pp. 371-390.

[19] B. Shirazi, M. Wang, and G. Pathak, “Analysis and Evaluation of Heuristic Methods for Static Scheduling,” *Journal of Parallel and Distributed Computing*, no. 10, 1990, pp. 222-232.