Exploiting Duplication to Minimize the Execution Times of Parallel Programs on Message-Passing Systems

Yu-Kwong Kwok and Ishfaq Ahmad

Department of Computer Science
Hong Kong University of Science and Technology, Hong Kong

Abstract[1]

Communication overhead is one of the main factors that can limit the speedup of parallel programs on message-passing parallel architectures. This limiting factor is more predominant in distributed systems such as clusters of homogeneous or heterogeneous workstations. However, excessive communication overhead can be reduced by redundantly executing some of the tasks of a parallel program on which other tasks critically depend. In this paper, we study the problem of duplication-based static scheduling of parallel programs on parallel and distributed systems. Previous duplication-based scheduling algorithms assumed the availability of unlimited number of homogeneous processors. In this paper, we consider more practical scenarios: when the number of processors is limited, and when the system consists of heterogeneous computers. For the first scenario, we propose an algorithm which minimizes the execution of a parallel program by controlling the level of duplication according to the number of processors available. For the second scenario, we design an algorithm which simultaneously exploits duplication and processor heterogeneity to minimize the total execution time of a parallel program. The proposed algorithms are suitable for low as well as high communication-to-computation ratios.

1 Introduction

During the past few years, we have witnessed a spectacular growth of parallel and distributed systems. This is because a variety of innovative architectures have been designed exploiting advancements in processor technology, low overhead switches, fast communication channels, and rich interconnection network topologies. Despite these advances, most existing parallel computers suffer from the problem of excessive inter-processor communication overhead. The problem is even more severe in distributed systems where multiple machines physically located at different sites are used as a single virtual parallel machine. The individual machines in the system can be homogeneous or heterogeneous. An example of this kind of hardware platform is a collection of networked workstations also called *workstation farms* or *clusters*. Slower interprocessor communication in parallel computers and networked distributed systems can be a major performance degradation factor for programs involving frequent message-passing between different program modules. One way to alleviate this problem is to employ task duplication in parallel program scheduling. Task duplication means scheduling a parallel program by redundantly allocating some of its tasks on which other tasks of the program critically depend. This reduces the start times of waiting tasks and eventually improves the overall execution time of the entire program.

It is well known that the multiprocessor scheduling problem in its many variants is NP-complete [4], [5], [6]. Practical solutions are mostly based on heuristics [12], [13], [14]. Many efficient scheduling algorithms employ a two-phase approach [1], [7], [8], [15]. In the first phase, priorities are assigned to the tasks. In the second phase, tasks are scheduled to appropriate processors one after the other according to their priorities. Various methods are used to determine task priorities. It has been shown that priorities based on the *critical path* can generate near-optimal schedules [9]. The critical path of a parallel program is an important attribute because it determines the lower bound on the overall execution time of the program. 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 efficiently by identifying the critical tasks and then redundantly allocating them in those slots, the overall execution time of the program can be further reduced. However, using task duplication makes the scheduling problem more complicated. 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.

In this paper, we study the problem of scheduling parallel programs using task duplication for homogeneous and heterogeneous message-passing multicomputers. Duplication based scheduling is relatively less explored with a few exceptions [3], [10]. We have proposed an algorithm which outperforms two previously reported algorithms using task duplication [2]. However, the previous algorithms as well as our new algorithm assume the availability of unlimited processors. To relieve this

1. This research was supported by Hong Kong Research Grants Council under contract number HKUST179/93E.

0-8186-6427-4/94 $04.00

constraint and consider duplication-based scheduling for more practical cases, we propose an algorithm which assumes limited number of processors and takes this number as an input parameter. The algorithm is self-adjusting in that it controls the level of duplication according to the number of available processors. We also design an algorithm for heterogeneous processors systems. This algorithm simultaneously exploits duplication and heterogeneity to minimize the execution time of a parallel program. The level of heterogeneity indicates the relative variation in the processing power of available machines. The proposed algorithm exploits this heterogeneity by scheduling the longer scheduling sequences on faster processors. We have used a number of task graphs including a number of parallel algorithms such as Gaussian elimination, FFT, LU-decomposition, Laplace equation solver, and various other syntactic graphs such as tree, fork-joins and completely random graphs, to test the performance of the proposed algorithms.

This paper is organized as follows. In Section 2, we give an overview of two recently reported and our previously proposed duplication-based scheduling algorithms, when the number of processors is unlimited. In Section 3, we present our new scheduling algorithms and discuss some of the principles used in their design. An example is used in illustrating the application of these algorithms. In Section 4, we first describe the workload used in testing the performance of our algorithms, and then present experimental results and performance comparisons. The last section contains the concluding remarks.

2 Related Work

Using task duplication in static scheduling is a relatively unexplored research problem and only a few scheduling algorithms have been proposed. One such algorithm, called *Duplication Scheduling Heuristic* (DSH) has been proposed in [10]. Another algorithm, called *Bottom-up-Top-down Duplication Heuristic* (BTDH), has been proposed in [3]. The DSH algorithm tries to minimize the start time of each task by duplicating its parent tasks into the idle time gap between the ready time and the start time of the task. The BTDH algorithm is essentially an extension of the DSH algorithm. The main difference between them is that the BTDH algorithm does not stop duplicating the parent tasks of a task even if its start time is increased temporarily. Due to this difference, the BTDH algorithm can generate a better schedule compared to the DSH algorithm when a given parallel program's *communication-to-computation ratio* (CCR), defined as the average inter-task communication cost divided by the average computation cost, is very high (e.g., 100). We have recently proposed a task duplication based scheduling algorithm called the *Critical Path Fast Duplication* (CPFD) algorithm [2], which outperforms the DSH and the BTDH algorithms consistently by a considerable margin. The CPFD algorithm tries to exploit all available idle time slots on a processor for task duplication so as to minimize the start time of each task. In addition, the CPFD algorithm always selects the most important task for scheduling at each step so that task duplication can be very effective in reducing the overall execution time. However, all of these algorithms assume the availability of unlimited number of processors. Such an assumption makes the scheduling algorithm less complex because of fewer constraints and therefore makes it easier to produce good schedules. Furthermore, this assumption may not always hold for more practical cases. Thus, our objectives in this work are to propose an algorithm that can minimize the total execution time of a program by exploiting duplication despite only a limited number of homogeneous processors, and to propose an algorithm for heterogeneous distributed systems.

3 The Proposed Algorithms

In our study, we assume that the processor network is fully-connected, and communication can take place simultaneously with computation. We represent a parallel program by a directed acyclic graph where each node in the task graph, labelled by n_i, represents a task. We use $w(n_i)$ to denote the computation cost of the task. An edge in the task graph represents the data dependency between two tasks. We use c_{ij} to denote the communication cost of the edge. The intra-processor communication cost is assumed to be negligible. Thus, if two tasks are scheduled to the same processor, c_{ij} is equal to zero. A node without any parent is called an *entry* node, while a node without any child is called an *exit* node. In the subsequent discussion, we will use the term *node* to denote a parallel program task.

It is not possible to determine the start time of a node before determining the start times of its parent nodes. Even when all the parents nodes of a node have been scheduled, there is an additional 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: *Let $MINPE(n_{i_k})$ be the processor on which the k-th parent node n_{i_k} of n_i is scheduled at its earliest start time, then the communication-constrained earliest start time of n_i on a processor J, denoted by CEST(n_i, 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. The parent node that maximizes the above expression is called the Very-Important-Parent of n_i and is denoted by VIP(n_i).

Given the communication-constrained earliest start time of a task 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 tasks $\{n_{J_1}, n_{J_2}, \ldots, n_{J_m}\}$ has been scheduled if 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 it. 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 given by*

$$max\{FT(n_{J_l}, J), CEST(n_i, J)\}$$

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.

Scheduling of nodes requires an attribute for determining the priorities of nodes [8], [9], which is given by the following definitions.

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 which has the largest sum of computation and communication costs. A node on the CP is called a CPN (Critical Path Node).*

Definition 3: *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 CPN. An Out-Branch Node (OBN) is a node, which is neither a CPN nor an IBN.*

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.*

From the above definitions, it can be noted that all the IBNs of a CPN should be scheduled before the CPN itself is scheduled. After all the CPNs have been scheduled, each of the remaining OBNs can be scheduled according to the OBN binding.

For selecting a processor, the start time of a node on a processor is constrained by the communication with its parent nodes. However, communication delay may be reduced if we apply duplication to some parent nodes of the candidate node. The following theorem governs the duplication process in our proposed algorithms.

Theorem 1: *At a particular scheduling step, for any node n_i and processor J, if*

i) $EST(n_i, J) = CEST(n_i, J)$, *and*

ii) $EST(VIP(n_i, J), J) + w(VIP(n_i, J)) < EST(n_i, J)$,

EST(n_i, J) can be reduced by scheduling VIP(n_i) to processor J with ST(n_i, J) set to EST(VIP(n_i), J).

Proof: Let $n_p = VIP(n_i)$ and $K = MINPE(n_p)$. Condition i) implies

$$EST(n_i, J) = FT(n_p, K) + c_{pi}$$

But $K \neq J$; otherwise

$$EST(n_i, J) = FT(n_p, J) = EST(n_p, J) + w(n_p)$$

contradicting condition ii). So we have

$$EST(n_i, J) = FT(n_p, K) + c_{pi}$$

Scheduling n_p to processor J at time $EST(n_p, J)$, we have

$$FT(n_p, J) = EST(n_p, J) + w(n_p) < FT(n_p, K) + c_{pi}$$

Thus, Axiom II is not governed by n_p and *EST(n_i, J)* in turn is reduced.❑

3.1 Algorithm for Unlimited Number of Homogeneous Processors

In this section, we present the first algorithm, called the *Economical Critical Path Fast Duplication* (ECPFD) algorithm. The ECPFD algorithm assumes that the number of processor available is limited. The algorithm adjusts the degree of duplication according to the number of available processors. Clearly, scheduling under the constraint of limited available processors may lead to generating longer schedule lengths than scheduling given unlimited number of processors. Thus, the scheduling algorithm has to make careful decision as to which processor is to be selected to accommodate a node. The ECPFD algorithm is formalized below. It should be noted that the number of processors available in the underlying system is a parameter to the ECPFD algorithm.

The ECPFD Algorithm:

(1) Determine a CP of the task graph. Break ties by selecting the one with a larger sum of computation costs.

(2) For each CPN, recursively schedule all the IBNs reaching it, in decreasing order of data arrival time, to processors that give the smallest start times by using Theorem 1 to decide whether a parent node of a node should be duplicated. Then, schedule the CPN itself to a processor, among all available processors, that allows it to start at the earliest time. Repeat this step for the next CPN.

(3) Perform the OBN binding.

(4) Without using any duplication, schedule each OBN to an 'already-in-use' processor that gives the smallest sum of the start times of the OBN and its critical child, which is the node that has the heaviest communication, *provided* the schedule length does not increase. If this fails, employ the same duplication process for scheduling CPNs to minimize the start time of the OBN.

The ECPFD algorithm first schedules all the CPNs as well as the IBNs reaching them, which are the most important nodes, to processors that give the smallest start times. To make use of the available processors, the ECPFD algorithm first attempts to schedule the OBNs, which are relatively less important compared to the CPNs, to

processors already in use. No duplication is applied in order to leave more space for subsequent scheduling. The ECPFD algorithm, however, does not pack OBNs to processors blindly. It selects the one which will not cause an increase in schedule length in the next step by checking the potential start time of a "critical" child node as well. If no processor is suitable and there are still some unused processors, the ECPFD reverts to schedule the OBN with duplication. This is done to make an effective use of the new processor by making an OBN start as early as possible. On the other hand, if no new processor is available, ECPFD selects the one which gives minimum increase in schedule length. Since both dominant steps — step (2) and step (4) — requires $O(pem)$ time, where p is the number of processors available, m is the number of nodes in the task graph and e is the number of edges in the task graph, the complexity of the ECPFD algorithm is also $O(pem^2)$.

3.2 Algorithm for Heterogeneous Processors

If the architecture of the distributed system is heterogeneous, that is, some processors run faster than others, schedule lengths can be significantly reduced compared with scheduling on homogeneous systems by properly utilizing the faster and slower processors. One possible way is to schedule the CPNs on the fastest processor. In order to tackle the scheduling problem on such a system, we propose the second algorithm called *Heterogeneous Critical Path Fast Duplication* (HCPFD) algorithm.

To model a heterogeneous distributed system, a parameter called *variance factor (VF)*, which ranges from 0 to 1, is used. For a given program, suppose it takes T time units to finish execution on the processor whose speed is the average of all the processors. Then, on the fastest processor, the execution time is equal to $T \times (1 - VF)$ while on the slowest processor, it is equal to $T \times (1 + VF)$. The heterogeneous processor system is equivalent to a homogeneous processor system in terms of the cumulative processing power. We assume the computation costs of nodes are statically determined on the processor which has the average speed. The communication links in the system are assumed to be homogeneous.

The HCPFD algorithm is formalized below. This algorithm takes the number of processors and the *VF* as a parameters.

The HCPFD Algorithm:

(1) Determine a CP of the task graph. Break ties by selecting the one with a larger sum of computation costs.

(2) For each CPN, recursively schedule all the IBNs reaching it, in decreasing order of data arrival time, to processors that give the smallest *finish* times by using Theorem 1 to decide whether a parent node of a node should be duplicated. The available processors are examined in decreasing order of processing speed. Note that we have to use the parameter *VF* to compute the *finish* time of a node on a particular processor. Then, schedule the CPN itself to a processor, among all available processors, that allows it to finish at the earliest time. Repeat this step for the next CPN.

(3) Perform the OBN binding.

(4) Without using any duplication, schedule each OBN to the *fastest* 'already-in-use' processor that gives the smallest sum of the *finish* times of the OBN and its critical child *provided* the schedule length does not increase (the critical child is the node that has the heaviest communication link). If this fails, employ the same duplication process for scheduling CPNs to minimize the *finish* time of the OBN.

As the computation cost varies from processor to processor, it is re-computed when checking Axiom I and Theorem 1 for different processors. The processor which gives the earliest *finish time* is selected to accommodate a node. The set of processors for consideration includes the fastest processors that accommodate the parent nodes of a candidate node in addition to the one holding the *VIP* of the candidate node. The complexity of the HCPFD algorithm is also $O(pem^2)$.

3.3 Illustrative Examples

In this section, we present example schedules generated by the DSH, BTDH, CPFD algorithm, ECPFD and the HCPFD algorithms. We use a task graph which represents the macro data-flow task graph for the parallel Gaussian elimination algorithm [11], [15] written in SPMD style. The graph is shown in Figure 1(a). An optimal schedule without duplication, which is generated by hand, is shown in Figure 1(b). The schedule length is 330 time units. It can be seen that the nodes n_{10}, n_{11} and n_{15} cannot start earlier because they have to wait for the data from their parent nodes that are scheduled to different processors. If their parent nodes are properly duplicated, these nodes may start at the earliest possible time and the schedule length may in turn be reduced.

To illustrate the performance of the DSH algorithm and the BTDH algorithm on this example, the HLFET (Highest Level First with Estimated Times) [1] algorithm is used as the auxiliary algorithm for determining node priorities. The HLFET algorithm, which computes priority for a candidate node by calculating the largest sum of computation costs from an entry node to the candidate node, is shown to give better results [3]. We call the two algorithms DSH/HLFET and BTDH/HLFET to indicate that they use the HLFET algorithm to determine node priorities. For this example task graph, the DSH/HLFET algorithm and the BTDH/HLFET algorithm generate the same schedule which is depicted in Figure 2. Here, the duplicated nodes are shown in shaded color, and the communication edges among nodes are omitted for clarity. The schedule length is 320 time units. Both algorithms do not apply duplication properly in that n_{18} cannot start earlier because of the inappropriate scheduling of n_{15}. The schedule generated by the CPFD

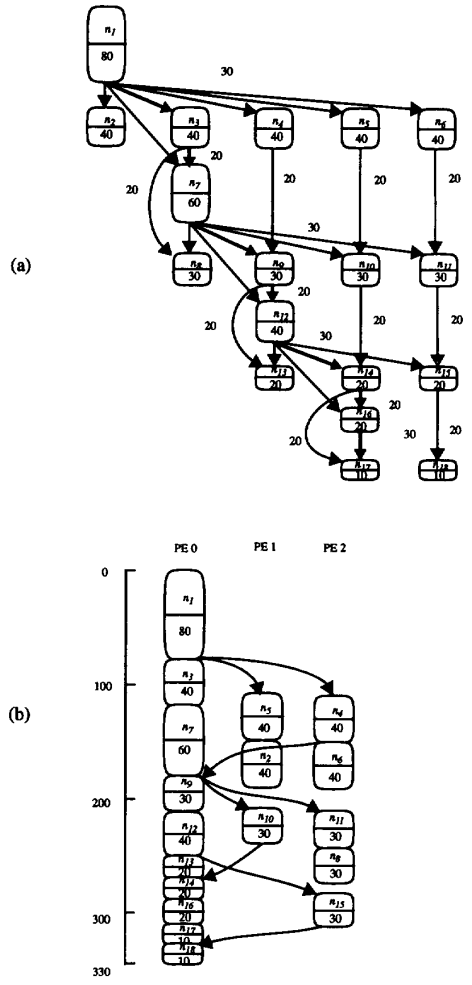

Figure 1: (a) A parallel Gaussian elimination task graph and (b) its optimal schedule without using duplication (schedule length = 330 time units).

algorithm is shown in Figure 3. Here, the schedule length is 300 time units which is the best possible schedule using duplication because all the CPNs start at their earliest possible start times. The schedule generated by the ECPFD algorithm is shown in Figure 4. Here, the schedule length is 310 time units but only 7 processors are used. The OBNs n_2, n_8 and n_{13} are properly scheduled to processors already in use. We also tested the ECPFD algorithm with 4 processors. The schedule length is then 320 time units (not shown here) which is the same as DSH and BTDH but at the expense of smaller number of processors. Assuming that the underlying system consists of 11 processors and the computation time variance factor is 0.5, the schedule generated by the HCPFD algorithm is shown in Figure 5. It

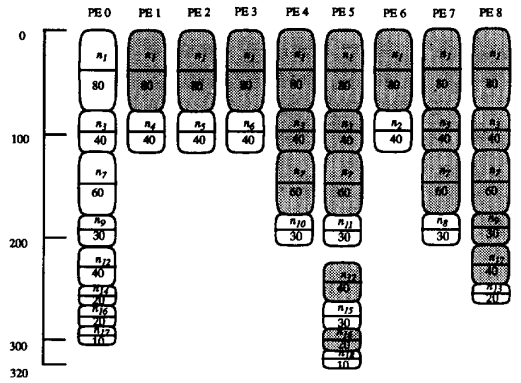

Figure 2: The schedule generated by the DSH/HLFET algorithm and the BTDH/HLFET algorithm (schedule length = 320 time units).

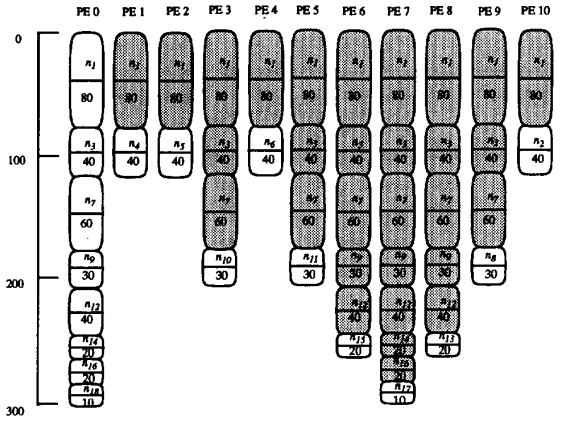

Figure 3: The schedule generated by the CPFD algorithm (schedule length = 300 time units).

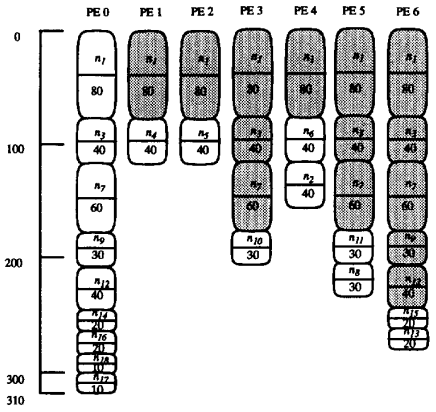

Figure 4: The schedule generated by the ECPFD algorithm (schedule length = 310 time units).

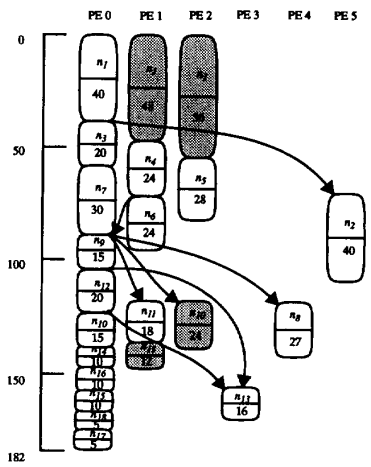

Figure 5: The schedule generated by the HCPFD algorithm onto heterogeneous processors with variance factor = 0.5 (schedule length = 182 time units).

can be seen that the computation costs on PE 0, which is the fastest processor, are half of that on a homogeneous processor. The schedule length (182 time units) is significantly smaller compared to the schedule generated by the CPFD algorithm which assumes a homogeneous system. Here, the HCPFD algorithm exploits the heterogeneity of the processors by proper scheduling of important nodes (CPNs) to the faster processors in the system.

4 Performance and Comparison

The proposed algorithms were simulated on a SUN SPARC IPX using various f task graphs. Our objective was to compare the schedule lengths produced by the three proposed algorithms for various graph structures, different values of CCR and the task graph size in terms of the number of nodes. For comparison, the DSH and BTDH algorithms were also simulated.

The suite of graphs consisted of 7 types of task graphs. The first type of task graphs represents the parallel Gaussian elimination algorithm. This task graph corresponds to the macro data-flow graphs for the this algorithm written in a SPMD style for distributed-memory multicomputers. The second type of task graphs represents the macro data-flow graph for the Laplace equation solver. The third type of graphs represents the LU-decomposition algorithms. In addition to these regular graphs which represent the three parallel algorithms, we generated some synthetic graphs whose structures are commonly encountered in various algorithms. One such graph is the in-tree graph in which each node has only one child. We generated this type of graphs by randomly selecting a number from a uniform distribution to be the number of levels in the tree. Given the total number of nodes to be generated, the number of nodes on each level was also randomly selected. Links between two adjacent levels or across levels were also randomly generated. The fifth type of graphs is the out-tree graph in which each node has only one parent. The sixth type of graphs is the fork-join graph which is a hybrid of an in-tree task graph and an out-tree task graph. In a fork-join task graph, there is a root node (with depth 0) which spawns a number of children. After the execution of all the children, the resulting data goes to the input of a single node which either spawns another set of children or terminates the whole parallel program. Finally, the seventh type consists of completely random graphs [15]. Within each type of graph structure, we used 7 values of CCR which are 0.1, 0.5, 1.0, 1.5, 2.0, 5.0 and 10.0. For each of these values, we generated 10 different graphs with the number of nodes varying from 10 to 100 with an increment of 10. Thus, for each type of graph structure and each value of CCR, 70 graphs were generated with the total number of graphs corresponding to 490. For each graph, the weights of the individual nodes and the communication edges are different and are randomly generated.

It has been shown in [2] that the performance of the CPFD algorithm is consistently superior than the other two algorithms. When CPFD is compared against DSH, it performs better for large and small values of CCRs. The average improvement in schedule lengths varies from 1.56% to 10.47%. The CPFD algorithm also outperforms the BTDH algorithm for all values of CCR. There is no single case out of 490 experiments, where CPFD perform worse than DSH or BTDH.

4.1 Limited Homogeneous Processors

To analyse the performance of the ECPFD algorithm which takes the number of processors as an input parameter, we made three comparisons between the ECPFD and the BTDH algorithms. We did not made the comparison of ECPFD and DSH since BTDH is already shown to be better than DSH [2]. For the first comparison, we observed the number of processors used by the BTDH algorithm for each task graph, and then used the same number of processors as the input to the ECPFD algorithm for that particular task graph. For the second comparison, we reduced the number of processors to 80%. In the third comparison, we used only 50% processors. The results of these experiments are given in Table I. For each comparison, there are seven rows in the table, with each row corresponding to results of running the scheduling algorithms on 70 tasks graphs for that value of CCR. The first columns indicate the comparative performance of the scheduling algorithms in terms of schedule lengths of these 70 task graphs. For example, when ECPD is compared with BTDH using the same number of processors for CCR equals 0.1, it generates

shorter schedules in 20 cases, generates longer schedules in 12 cases and generates the same schedule lengths in 38 cases. The next three columns indicate the average percentage improvement, maximum percentage improvement and average percentage degradation in the schedule lengths produced. In addition, for each comparison, the Table I provides the cumulative results for all values of CCR.

Table I: A performance comparison of ECPFD and BTDH with limited number of processors.

			Number of times it performs better	Number of times it performs worse	Number of times it performs the same	The average% improvement in schedule length	The maximum% improvement in schedule length	The worst % degradation in schedule length
ECPFD compared with BTDH (same number of processors)	CCR	0.1	20	12	38	1.12	16.51	3.96
		0.5	30	3	37	1.77	12.22	1.34
		1.0	40	0	30	2.73	12.59	0.00
		1.5	42	3	25	2.97	12.48	2.69
		2.0	35	5	30	2.23	11.98	12.42
		5.0	23	6	41	-0.21	10.12	44.57
		10.0	20	10	40	-0.99	10.56	37.62
		All	210	39	241	1.76	16.51	44.57
ECPFD compared with BTDH (ECPFD uses only 80% processors)	CCR	0.1	18	19	33	0.33	12.99	11.25
		0.5	25	7	38	0.89	11.72	28.68
		1.0	39	0	31	2.27	11.63	0.00
		1.5	39	7	24	1.59	12.48	41.29
		2.0	32	12	26	1.25	9.80	16.86
		5.0	22	21	27	-1.39	10.12	44.57
		10.0	17	18	35	-2.73	10.55	54.27
		All	192	84	214	0.32	12.99	54.27
ECPFD compared with BTDH (ECPFD uses only 50% processors)	CCR	0.1	11	49	10	-11.64	6.61	53.33
		0.5	9	49	12	-10.61	6.18	53.33
		1.0	21	36	13	-8.11	6.58	48.38
		1.5	18	39	13	-5.74	6.09	103.68
		2.0	17	44	9	-7.05	8.67	63.92
		5.0	13	48	9	-9.35	9.69	133.55
		10.0	11	51	8	-16.99	9.19	184.65
		All	100	316	74	-9.92	9.69	184.65

From the results of the first comparison, ECPFD collectively performs better than BTDH in 210 cases and performs worse in only 39 cases. There were a few cases in which ECPFD performed quite poorly. These cases accounted for the negative value of average percentage improvement when CCR was equal to 5.0 and 10.0. However, ECPFD performed better than BTDH for more number of times. In the second comparison, when 80% processors were used, ECPFD again outperforms BTDH for all values of CCR except 5.0 and 10.0. At these values of CCR, the performance of both algorithms is about the same. When the number of processors is reduced to 50%, ECPFD is still able to outperform BTDH in 100 cases. These results are better understood by analysing the efficiency which is defined as the serial time of the task graph divided by the parallel time (schedule length) divided by the number of processors used. The efficiency of BTDH and ECPFD with 100%, 80% and 50% processors is shown in Figure 6(a) and Figure 6(b) for varying number of nodes in a task graph and values of CCR, respectively. The efficiency of the CPFD algorithm is also included for comparison. These results for Figure 6(a) and Figure 6(b) were obtained by taking the averages across all cases by fixing the number of nodes and CCR, respectively. Figure 6(a) in a sense indicates the scalability of these algorithms which is an important measure of the parallelism captured by the scheduling algorithm. On the other hand, Figure 6(b) indicates the effect of CCR which is a crucial reason for using duplication in scheduling. The results of these figures indicate that the CPFD algorithm is slightly less efficient than the BTDH algorithm. However, as shown earlier, CPFD clearly performs better than BTDH if no limit on the number of processors is imposed. The ECPFD algorithm is clearly more efficient because it drastically reduces the number of processors at the expense of little degradation in schedule length. It also scales well with the graph size and remains more efficient at larger values of CCR. The choice of using CPFD or ECPFD obviously depends on the number of processors available.

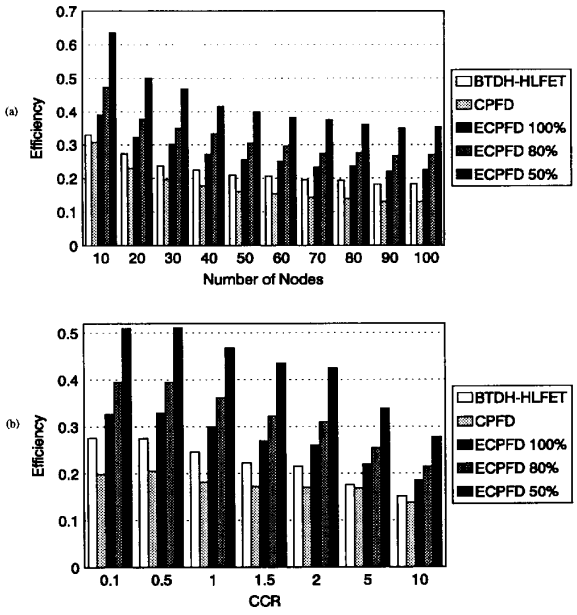

Figure 6: Efficiency of the BTDH, CFPD and ECPFD (100%, 80% and 50% processors) algorithms for various task graph sizes and CCRs.

4.2 Heterogeneous Processors

In this section, we present the results of the HCPFD algorithm on heterogeneous processors. The algorithm, as stated above, is designed to minimize the schedule length by taking advantage of heterogeneity. We used the same set of task graphs for testing the performance of the HCPFD algorithm. We used 4 values of the variance of heterogeneity which were 0.2, 0.3, 0.5 and 0.9. The results of our experiments are provided in Figure 7(a) and Figure 7(b) showing the percentage improvement in schedule length obtained with HCPFD over CPFD for different graph sizes and CCRs, respectively. These figures indicate that HCPFD exploits heterogeneity by yielding an improvement in schedule length. This is because HCFPD systematically schedules more important nodes to faster processors and less important nodes to slower processors. The charts shown in these figures indicate that better schedule length is obtained if the variance factor is large.

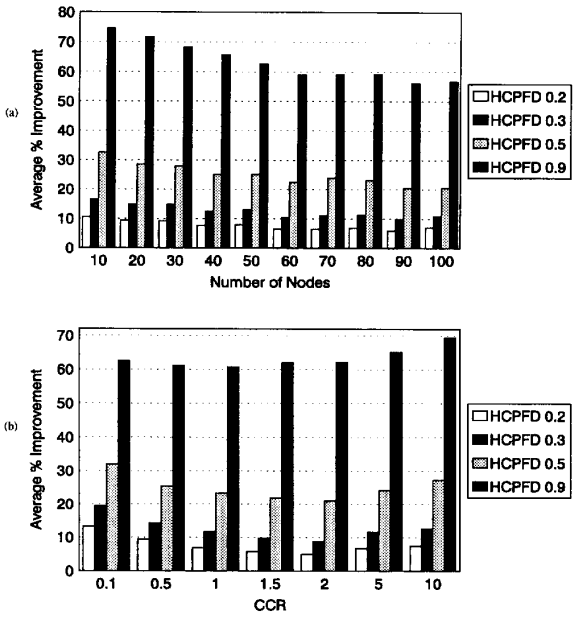

Figure 7: Percentage improvement in schedule length with HCPFD (VF = 0.2, 0.3, 0.5, 0.9) over CPFD at various task graph sizes and CCRs.

5 Conclusions

We have proposed two scheduling algorithms using task duplication technique. These scheduling algorithms can be particularly useful when the CCR of a parallel algorithm on a given system is high. The technique used in our proposed algorithms is to systematically decompose the task graph into CP, IBN and OBN bindings. These bindings help the algorithms to first identify the relative importance of nodes and then, according to their importance, enable them to start them at their earliest possible start times. The ECPFD algorithm is designed to control the degree of duplication according to the number of available processors. The ECPFD algorithm is more efficient than previously proposed algorithms because it uses less number of processors. The HCPFD algorithm which is designed to exploit the heterogeneity of the processors can be useful for distributed systems.

References

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

[2] I. Ahmad and Y.K. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," to appear in *Proc. Int'l Conf. on Parallel Processing*, Aug. 1994.

[3] 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*, pp. 512-521, Nov. 1992.

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

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

[6] M.R. Gary and D.S. Johnson, *Computers and Intractability*, W.H. Freeman and Company, 1979.

[7] 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.

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

[9] W.H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on MultiProcessor Systems," *IEEE Trans. on Computers*, vol. C-24, pp. 1235-1238, Dec. 1975.

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

[11] R.E. Lord, J.S. Kowalik and S.P. Kumar, "Solving Linear Algebraic Equations on an MIMD Computer," *Journal of the ACM*, 30(1), pp. 103-117, Jan. 1983.

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

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

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

[15] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, Jul. 1990.