Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing

Haluk Topcuoglu, *Member*, *IEEE*, Salim Hariri, *Member*, *IEEE Computer Society*, and Min-You Wu, *Senior Member*, *IEEE*

Abstract—Efficient application scheduling is critical for achieving high performance in heterogeneous computing environments. The application scheduling problem has been shown to be NP-complete in general cases as well as in several restricted cases. Because of its key importance, this problem has been extensively studied and various algorithms have been proposed in the literature which are mainly for systems with homogeneous processors. Although there are a few algorithms in the literature for heterogeneous processors, they usually require significantly high scheduling costs and they may not deliver good quality schedules with lower costs. In this paper, we present two novel scheduling algorithms for a bounded number of heterogeneous processors with an objective to simultaneously meet high performance and fast scheduling time, which are called the Heterogeneous Earliest-Finish-Time (HEFT) algorithm and the Critical-Path-on-a-Processor (CPOP) algorithm. The HEFT algorithm selects the task with the highest upward rank value at each step and assigns the selected task to the processor, which minimizes its earliest finish time with an insertion-based approach. On the other hand, the CPOP algorithm uses the summation of upward and downward rank values for prioritizing tasks. Another difference is in the processor selection phase, which schedules the critical tasks onto the processor that minimizes the total execution time of the critical tasks. In order to provide a robust and unbiased comparison with the related work, a parametric graph generator was designed to generate weighted directed acyclic graphs with various characteristics. The comparison study, based on both randomly generated graphs and the graphs of some real applications, shows that our scheduling algorithms significantly surpass previous approaches in terms of both quality and cost of schedules, which are mainly presented with schedule length ratio, speedup, frequency of best results, and average scheduling time metrics.

Index Terms—DAG scheduling, task graphs, heterogeneous systems, list scheduling, mapping.

1 Introduction

Diverse sets of resources interconnected with a high-speed network provide a new computing platform, called the heterogeneous computing system, which can support executing computationally intensive parallel and distributed applications. A heterogeneous computing system requires compile-time and runtime support for executing applications. The efficient scheduling of the tasks of an application on the available resources is one of the key factors for achieving high performance.

The general task scheduling problem includes the problem of assigning the tasks of an application to suitable processors and the problem of ordering task executions on each resource. When the characteristics of an application which includes execution times of tasks, the data size of communication between tasks, and task dependencies are known a priori, it is represented with a *static* model.

In the general form of a static task scheduling problem, an application represented by a directed acyclic graph (DAG) in which nodes represent application tasks and edges represent intertask data dependencies. Each node label shows computation cost (expected computation time) of the task and each edge label shows intertask communication cost (expected communication time) between tasks. The objective function of this problem is to map tasks onto processors and order their executions so that task-precedence requirements are satisfied and a minimum overall completion time is obtained. The task scheduling problem is NP-complete in the general case [1], as well as some restricted cases [2], such as scheduling tasks with one or two time units to two processors and scheduling unit-time tasks to an arbitrary number of processors.

Because of its key importance on performance, the task scheduling problem in general has been extensively studied and various heuristics were proposed in the literature [3], [4], [5], [6], [7], [8], [9], [10], [11], [13], [12], [16], [17], [18], [20], [22], [23], [27], [30]. These heuristics are classified into a variety of categories (such as list-scheduling algorithms, clustering algorithms, duplication-based algorithm, guided random search methods) and they are mainly for systems with homogeneous processors.

In a list scheduling algorithm [3], [4], [6], [7], [18], [22], an ordered list of tasks is constructed by assigning priority for each task. Tasks are selected in the order of their priorities and each selected task is scheduled to a processor which minimizes a predefined cost function. The algorithms in this category provide good quality of schedules and their performance is comparable with the other categories at a

- *H. Topcuoglu is with the Computer Engineering Department, Marmara University, Goztepe Kampusu, 81040, Istanbul, Turkey. E-mail: haluk@eng.marmara.edu.tr.*
- *S. Hariri is with the Department of Electrical and Computer Engineering, University of Arizona, Tucson, AZ 85721-0104. E-mail: hariri@ece.arizona.edu.*
- *M.-Y. Wu is with the Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131-1356. E-mail: wu@eece.unm.edu.*

Manuscript received 28 Aug. 2000; revised 12 July 2001; accepted 6 Sept. 2001.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 112783.

lower scheduling time [21], [26]. The clustering algorithms [3], [12], [19], [25] are, in general for an unbounded number of processors, so they may not be directly applicable. A clustering algorithm requires a second phase (a scheduling module) to merge the task clusters generated by the algorithm onto a bounded number of processors and to order the task executions within each processor [24]. Similarly, task duplication-based heuristics are not practical because of their significantly high time complexity. As an example, the time complexity of the BTDH Algorithm [30] and the DSH Algorithm [18] are $O(v^4)$; the complexity of the CPFD Algorithm [9] is $O(e \times v^2)$ for scheduling v tasks connected with e edges on a set of homogeneous processors.

Genetic Algorithms [5], [8], [11], [13], [17], [31] (GAs) are of the most widely studied guided random search techniques for the task scheduling problem. Although they provide good quality of schedules, their execution times are significantly higher than the other alternatives. It was shown that the improvement of the GA-based solution to the second best solution was not more than 10 percent and the GA-based approach required around a minute to produce a solution, while the other heuristics required an execution of a few seconds [31]. Additionally, extensive tests are required to find optimal values for the set of control parameters used in GA-based solutions.

The task scheduling problem has also been studied by a few research groups for the heterogeneous systems [6], [7], [8], [10], [11], [13], [14]. These algorithms may require assigning a set of control parameters and some of them confront with the substantially high scheduling costs [6], [8], [11], [13]. Some of them partition the tasks in a DAG into levels such that there will be no dependency between tasks in the same level [10], [14]. This level-by-level scheduling technique considers the tasks only in the current level (that is, a subset of ready tasks) at any time, which may not perform well because of not considering all ready tasks. Additionally, the study given in [14] presents a dynamic remapper that requires an initial schedule of a given DAG and then improves its performance using three variants of an algorithm, which is out of the scope of this paper.

In this paper, we propose two new static scheduling algorithms for a bounded number of fully connected heterogeneous processors: the Heterogeneous Earliest-Finish-Time (HEFT) algorithm and the Critical-Path-on-a-Processor (CPOP) algorithm. Although the static-scheduling for heterogeneous systems is offline, in order to provide a practical solution, the scheduling time (or running time) of an algorithm is the key constraint. Therefore, the motivation behind these algorithms is to deliver good-quality of schedules (or outputs with better scheduling lengths) with lower costs (i.e., lower scheduling times). The HEFT Algorithm selects the task with the highest *upward rank* (defined in Section 4.1) at each step. The selected task is then assigned to the processor which minimizes its earliest finish time with an insertion-based approach. The upward rank of a task is the length of the critical path (i.e., the longest path) from the task to an exit task, including the computation cost of the task. The CPOP algorithm selects the task with the highest (upward rank + downward rank) value at each step. The algorithm targets scheduling of all critical tasks (i.e., tasks on the critical path of the DAG) onto a single processor, which minimizes the total execution time of the critical tasks. If the selected task is noncritical, the processors selection phase is based on earliest execution time with insertion-based scheduling, as in the HEFT Algorithm.

As part of this research work, a parametric graph generator has been designed to generate weighted directed acyclic graphs for the performance study of the scheduling algorithms. The graph generator targets the generation of many types of DAGs using several input parameters that provide an unbiased comparison of task-scheduling algorithms. The comparison study in this paper is based on both randomly generated task graphs and the task graphs of real applications, including the Gaussian Elimination Algorithm [3], [28], FFT Algorithm [29], [30], and a molecular dynamic code given in [19]. The comparison study shows that our algorithms significantly surpass previous approaches in terms of both performance metrics (schedule length ratio, speedup, efficiency, and number of occurrences giving best results) and a cost metric (scheduling time to deliver an output schedule).

The remainder of this paper is organized as follows: In the next section, we define the research problem and the related terminology. In Section 3, we provide a taxonomy of task-scheduling algorithms and the related work in scheduling for heterogeneous systems. Section 4 introduces our scheduling algorithms (the HEFT and the CPOP Algorithms). Section 5 presents a comparison study of our algorithms with the related work, which is based on randomly generated task graphs and task graphs of several real applications. In Section 6, we introduce several extensions to the HEFT algorithm. The summary of the research presented and planned future work is given in Section 7.

2 Task-Scheduling Problem

A scheduling system model consists of an application, a target computing environment, and a performance criteria for scheduling. An *application* is represented by a directed acyclic graph, $G = (V, E)$, where V is the set of v tasks and E is the set of e edges between the tasks. (*Task* and *node* terms are interchangeably used in the paper.) Each edge $(i, j) \in E$ represents the precedence constraint such that task n_i should complete its execution before task n_j starts. $Data$ is a $v \times v$ matrix of communication data, where $data_{i,k}$ is the amount of data required to be transmitted from task n_i to task n_k.

In a given task graph, a task without any parent is called an *entry task* and a task without any child is called an *exit task*. Some of the task scheduling algorithms may require single-entry and single-exit task graphs. If there is more than one exit (entry) task, they are connected to a zero-cost *pseudo exit (entry) task* with zero-cost edges, which does not affect the schedule.

We assume that the target computing environment consists of a set Q of q heterogeneous processors connected in a fully connected topology in which all interprocessor communications are assumed to perform without contention. In our model, it is also assumed that

computation can be overlapped with communication. Additionally, task executions of a given application are assumed to be nonpreemptive. W is a $v \times q$ computation cost matrix in which each $w_{i,j}$ gives the estimated execution time to complete task n_i on processor p_j. Before scheduling, the tasks are labeled with the average execution costs. The average execution cost of a task n_i is defined as

$$\overline{w_i} = \sum_{j=1}^{q} w_{i,j}/q. \tag{1}$$

The data transfer rates between processors are stored in matrix B of size $q \times q$. The communication startup costs of processors are given in a q-dimensional vector L. The communication cost of the edge (i,k), which is for transferring data from task n_i (scheduled on p_m) to task n_k (scheduled on p_n), is defined by

$$c_{i,k} = L_m + \frac{data_{i,k}}{B_{m,n}}. \tag{2}$$

When both n_i and n_k are scheduled on the same processor, $c_{i,k}$ becomes zero since we assume that the intraprocessor communication cost is negligible when it is compared with the interprocessor communication cost. Before scheduling, average communication costs are used to label the edges. The average communication cost of an edge (i,k) is defined by

$$\overline{c_{i,k}} = \overline{L} + \frac{data_{i,k}}{\overline{B}}, \tag{3}$$

where $\overline{B}$ is the average transfer rate among the processors in the domain and $\overline{L}$ is the average communication startup time.

Before presenting the objective function, it is necessary to define the EST and EFT attributes, which are derived from a given partial schedule. $EST(n_i, p_j)$ and $EFT(n_i, p_j)$ are the *earliest* execution start time and the *earliest* execution finish time of task n_i on processor p_j, respectively. For the entry task n_{entry},

$$EST(n_{entry}, p_j) = 0. \tag{4}$$

For the other tasks in the graph, the EFT and EST values are computed recursively, starting from the entry task, as shown in (5) and (6), respectively. In order to compute the EFT of a task n_i, all immediate predecessor tasks of n_i must have been scheduled.

$$EST(n_i, p_j) = \max\left\{avail[j], \max_{n_m \in pred(n_i)} (AFT(n_m) + c_{m,i})\right\}, \tag{5}$$

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j)\ , \tag{6}$$

where $pred(n_i)$ is the set of immediate predecessor tasks of task n_i and $avail[j]$ is the earliest time at which processor p_j is ready for task execution. If n_k is the last assigned task on processor p_j, then $avail[j]$ is the time that processor p_j completed the execution of the task n_k and it is ready to execute another task when we have a noninsertion-based scheduling policy. The inner *max* block in the EST equation returns the *ready time*, i.e., the time when all data needed by n_i has arrived at processor p_j.

After a task n_m is scheduled on a processor p_j, the earliest start time and the earliest finish time of n_m on processor p_j is equal to the actual start time, $AST(n_m)$, and the actual finish time, $AFT(n_m)$, of task n_m, respectively. After all tasks in a graph are scheduled, the schedule length (i.e., overall completion time) will be the actual finish time of the exit task n_{exit}. If there are multiple exit tasks and the convention of inserting a pseudo exit task is not applied, the schedule length (which is also called *makespan*) is defined as

$$makespan\ =\ \max\{AFT(n_{exit})\}. \tag{7}$$

The *objective function* of the task-scheduling problem is to determine the assignment of tasks of a given application to processors such that its schedule length is minimized.

3 Related Work

Static task-scheduling algorithms can be classified into two main groups (see Fig. 1), heuristic-based and guided random-search-based algorithms. The former can be further classified into three groups: list scheduling heuristics, clustering heuristics, and task duplication heuristics.

List Scheduling Heuristics. A list-scheduling heuristic maintains a list of all tasks of a given graph according to their priorities. It has two phases: the *task prioritizing* (or *task selection*) phase for selecting the highest-priority ready task and the *processor selection* phase for selecting a suitable processor that minimizes a predefined cost function (which can be the execution start time). Some of the examples are the Modified Critical Path (MCP) [3], Dynamic Level Scheduling [6], Mapping Heuristic (MH) [7], Insertion-Scheduling Heuristic [18], Earliest Time First (ETF) [22], and Dynamic Critical Path (DCP) [4] algorithms. Most of the list-scheduling algorithms are for a bounded number of fully connected homogeneous processors. List-scheduling heuristics are generally more practical and provide better performance results at a lower scheduling time than the other groups.

Clustering Heuristics. An algorithm in this group maps the tasks in a given graph to an unlimited number of clusters. At each step, the selected tasks for clustering can be any task, not necessarily a ready task. Each iteration refines the previous clustering by merging some clusters. If two tasks are assigned to the same cluster, they will be executed on the same processor. A clustering heuristic requires additional steps to generate a final schedule: a cluster merging step for merging the clusters so that the remaining number of clusters equal the number of processors, a cluster mapping step for mapping the clusters on the available processors, and a task ordering step for ordering the mapped tasks within each processor [24]. Some examples in this group are the Dominant Sequence Clustering (DSC) [12], Linear Clustering Method [19], Mobility Directed [3], and Clustering and Scheduling System (CASS) [25].

Task Duplication Heuristics. The idea behind duplication-based scheduling algorithms is to schedule a task graph by mapping some of its tasks redundantly, which reduces the interprocess communication overhead [9], [18], [27], [30]. Duplication-based algorithms differ according to the selection strategy of the tasks for duplication. The

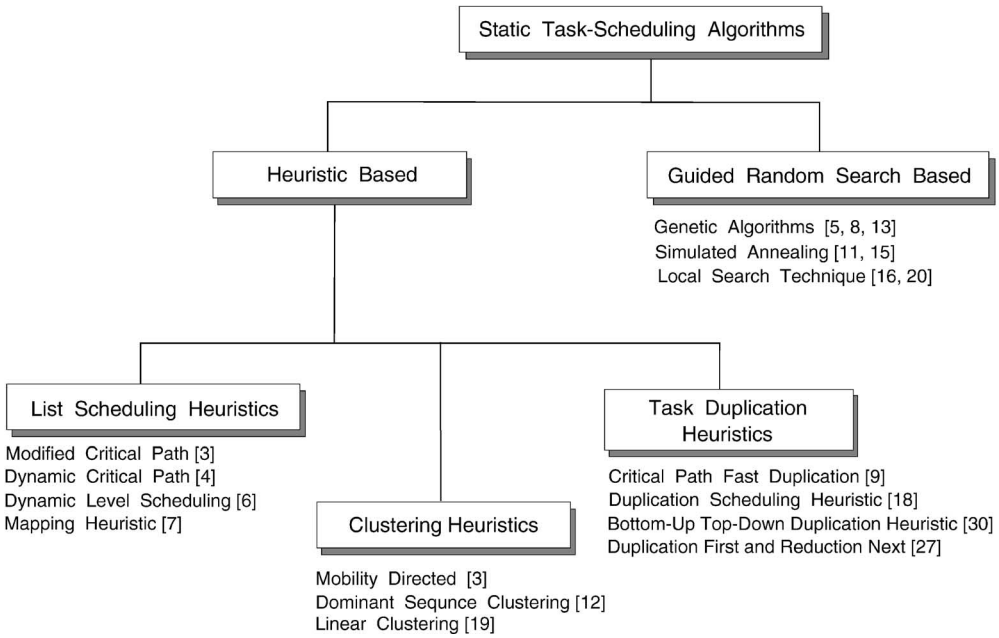


Fig. 1. Classification of static task-scheduling algorithms.

algorithms in this group are usually for an unbounded number of identical processors and they have much higher complexity values than the algorithms in the other groups.

Guided Random Search Techniques. Guided random search techniques (or randomized search techniques) use random choice to guide themselves through the problem space, which is not the same as performing merely random walks as in the random search methods. These techniques combine the knowledge gained from previous search results with some randomizing features to generate new results. Genetic algorithms (GAs) [5], [8], [11], [13], [17] are the most popular and widely used techniques for several flavors of the task scheduling problem. GAs generate good quality of output schedules; however, their scheduling times are usually much higher than the heuristic-based techniques [31]. Additionally, several control parameters in a genetic algorithm should be determined appropriately. The optimal set of control parameters used for scheduling a task graph may not give the best results for another task graph. In addition to GAs, simulated annealing [11], [15] and local search method [16], [20] are the other methods in this group.

3.1 Task-Scheduling Heuristics for Heterogeneous Environments

This section presents the reported task-scheduling heuristics that support heterogeneous processors, which are the Dynamic Level Scheduling Algorithm [6], the Levelized-Min Time Algorithm [10], and the Mapping Heuristic Algorithm [7].

Dynamic-Level Scheduling (DLS) Algorithm. At each step, the algorithm selects the (ready node, available processor) pair that maximizes the value of the *dynamic level* which is equal to $DL(n_i, p_j) = rank_u^s(n_i) - EST(n_i, p_j)$. The computation cost of a task is the median value of the computation costs of the task on the processors. In this algorithm, upward rank calculation does not consider the communication costs. For heterogeneous environments, a new term added for the difference between the task's median execution time on all processors and its execution time on the current processor. The general DLS algorithm has an $O(v^3 \times q)$ time complexity, where v is the number of tasks and q is the number of processors.

Mapping Heuristic (MH). In this algorithm, the computation cost of a task on a processor is computed by the number of instructions to be executed in the task divided by the speed of the processor. However, in setting the computation costs of tasks and the communication costs of edges before scheduling, similar processing elements (i.e., homogeneous processors) are assumed; the heterogeneity comes into the picture during the scheduling process.

This algorithm uses *static* upward ranks to assign priorities. (The authors also experimented by adding the communication delay to the rank values.) In this algorithm, the *ready time* of a processor for a task is the time when the processor has finished its last assigned task and is ready to execute a new one. The MH algorithm does not schedule a task to an idle time slot that is between two tasks already scheduled. The time complexity, when contention is considered, is equal to $O(v^2 \times q^3)$ for v tasks and q processors; otherwise, it is equal to $O(v^2 \times q)$.

Levelized-Min Time (LMT) Algorithm. It is a two-phase algorithm. The first phase groups the tasks that can be executed in parallel using the *level* attribute. The second phase assigns each task to the fastest available processor. A task in a lower level has higher priority than a task in a higher level. Within the same level, the task with the highest computation cost has the highest priority. Each task is assigned to a processor that minimizes the sum of the task's computation cost and the total communication costs with tasks in the previous levels. For a fully connected graph, the time complexity is $O(v^2 \times q^2)$ when there are v tasks and q processors.

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task.
3. Sort the tasks in a scheduling list by nonincreasing order of $rank_u$ values.
4. **while** there are unscheduled tasks in the list **do**
5. Select the first task, n_i, from the list for scheduling.
6. **for** each processor p_k in the processor-set ($p_k \in Q$) **do**
7. Compute $EFT(n_i, p_k)$ value using the *insertion-based scheduling* policy.
8. Assign task n_i to the processor p_j that minimizes EFT of task n_i.
9. **endwhile**

Fig. 2. The HEFT algorithm.

4 Task-Scheduling Algorithms

Before introducing the details of HEFT and CPOP algorithms, we introduce the graph attributes used for setting the task priorities.

4.1 Graph Attributes Used by HEFT and CPOP Algorithms

Tasks are ordered in our algorithms by their scheduling priorities that are based on upward and downward ranking. The *upward rank* of a task n_i is recursively defined by

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)), \qquad (8)$$

where $succ(n_i)$ is the set of immediate successors of task n_i, $\overline{c_{i,j}}$ is the average communication cost of edge (i, j), and $\overline{w_i}$ is the average computation cost of task n_i. Since the rank is computed recursively by traversing the task graph upward, starting from the exit task, it is called *upward* rank. For the exit task n_{exit}, the upward rank value is equal to

$$rank_u(n_{exit}) = \overline{w_{exit}}. \qquad (9)$$

Basically, $rank_u(n_i)$ is the length of the critical path from task n_i to the exit task, including the computation cost of task n_i. There are algorithms in the literature which compute the rank value using computation costs only, which is called *static* upward rank, $rank_u^s$.

Similarly, the *downward rank* of a task n_i is recursively defined by

$$rank_d(n_i) = \max_{n_j \in pred(n_i)} \{rank_d(n_j) + \overline{w_j} + \overline{c_{j,i}}\}, \qquad (10)$$

where $pred(n_i)$ is the set of immediate predecessors of task n_i. The downward ranks are computed recursively by traversing the task graph downward starting from the entry task of the graph. For the entry task n_{entry}, the downward rank value is equal to zero. Basically, $rank_d(n_i)$ is the longest distance from the entry task to task n_i, excluding the computation cost of the task itself.

4.2 The Heterogeneous-Earliest-Finish-Time (HEFT) Algorithm

The HEFT algorithm (Fig. 2) is an application scheduling algorithm for a bounded number of heterogeneous processors, which has two major phases: a *task prioritizing phase* for computing the priorities of all tasks and a *processor selection phase* for selecting the tasks in the order of their priorities and scheduling each selected task on its "best" processor, which minimizes the task's finish time.

Task Prioritizing Phase. This phase requires the priority of each task to be set with the upward rank value, $rank_u$, which is based on mean computation and mean communication costs. The task list is generated by sorting the tasks by decreasing order of $rank_u$. Tie-breaking is done randomly. There can be alternative policies for tie-breaking, such as selecting the task whose immediate successor task(s) has higher upward ranks. Since these alternate policies increase the time complexity, we prefer a random selection strategy. It can be easily shown that the decreasing order of $rank_u$ values provides a topological order of tasks, which is a linear order that preserve the precedence constraints.

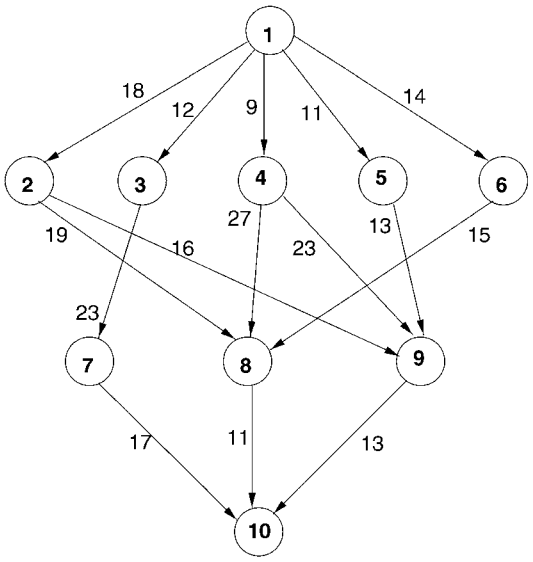
Processor Selection Phase. For most of the task scheduling algorithms, the earliest available time of a processor p_j for a task execution is the time when p_j completes the execution of its last assigned task. However, the HEFT algorithm has an insertion-based policy which considers the possible insertion of a task in an earliest idle time slot between two already-scheduled tasks on a processor. The length of an idle time-slot, i.e., the difference between execution start time and finish time of two tasks that were consecutively scheduled on the same processor, should be at least capable of computation cost of the task to be scheduled. Additionally, scheduling on this idle time slot should preserve precedence constraints.

In the HEFT Algorithm, the search of an appropriate idle time slot of a task n_i on a processor p_j starts at the time equal to the $ready\_time$ of n_i on p_j, i.e., the time when all input data of n_i that were sent by n_i's immediate predecessor tasks have arrived at processor p_j. The search continues until finding the first idle time slot that is capable of holding the computation cost of task n_i. The HEFT algorithm has an $O(e \times q)$ time complexity for e edges and q processors. For a dense graph when the number of edges is proportional to $O(v^2)$ (v is the number of tasks), the time complexity is on the order of $O(v^2 \times p)$.

As an illustration, Fig. 4a presents the schedules obtained by the HEFT algorithm for the sample DAG of Fig. 3. The schedule length, which is equal to 80, is shorter than the schedule lengths of the related work; specifically, the schedule lengths of DLS, MH, and LMT Algorithms are 91, 91, and 95, respectively. The first column in Table 1 gives upward rank values for the given task graph. The scheduling order of the tasks with respect to the HEFT Algorithm is $\{n_1, n_3, n_4, n_2, n_5, n_6, n_9, n_7, n_8, n_{10}\}$.

4.3 The Critical-Path-on-a-Processor (CPOP) Algorithm

Although our second algorithm, the CPOP algorithm shown in Fig. 5, has the task prioritizing and processor



Computation Costs

Task	P1	P2	P3
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

Fig. 3. A sample task graph with 10 tasks.

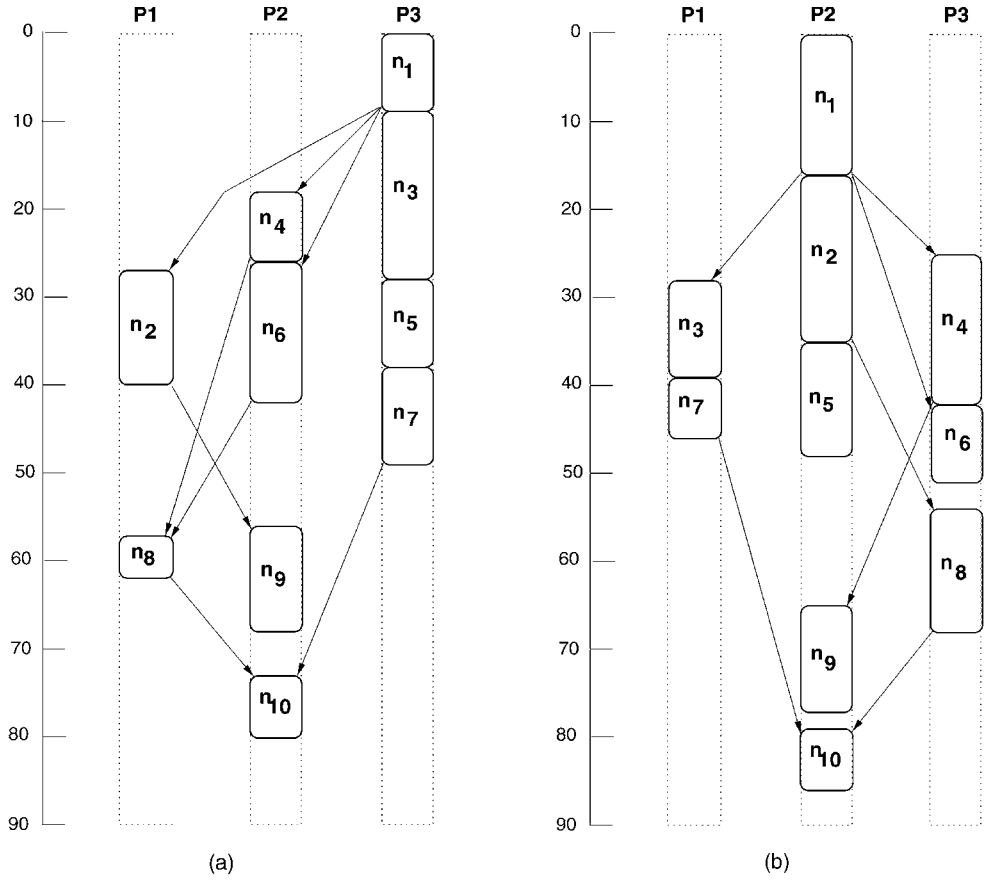


Fig. 4. Scheduling of task graph in Fig. 3 with the HEFT and CPOP algorithms. (a) HEFT Algorithm (schedule length = 80). (b) CPOP Algorithm (schedule length = 86).

selection phases as in the HEFT algorithm, it uses a different attribute for setting the task priorities and a different strategy for determining the "best" processor for each selected task.

Task Prioritizing Phase. In this phase, upward rank ($rank_u$) and downward rank ($rank_d$) values for all tasks are computed using mean computation and mean communication costs (Steps 1-3). The CPOP algorithm uses the critical path of a given application graph. The length of this path, $|CP|$, is the sum of the computation costs of the tasks on the path and intertask communication costs along the path. The sum of computation costs on the critical path of a graph is basically the *lower bound* for the schedule lengths generated by the task scheduling algorithms.

The priority of each task is assigned with the summation of upward and downward ranks. The critical path length is

TABLE 1
Values of Attributes Used in HEFT and CPOP Algorithms for Task Graph in Fig. 3

n_i	$rank_u(n_i)$	$rank_d(n_i)$	$rank_u(n_i) + rank_d(n_i)$
n_1	108.000	0.000	108.000
n_2	77.000	31.000	108.000
n_3	80.000	25.000	105.000
n_4	80.000	22.000	102.000
n_5	69.000	24.000	93.000
n_6	63.333	27.000	90.333
n_7	42.667	62.333	105.000
n_8	35.667	66.667	102.334
n_9	44.333	63.667	108.000
n_{10}	14.667	93.333	108.000

equal to the entry task's priority (Step 5). Initially, the entry task is the selected task and marked as a critical path task. An immediate successor (of the selected task) that has the highest priority value is selected and it is marked as a critical path task. This process is repeated until the exit node is reached (Steps 6-12). For tie-breaking, the first immediate successor which has the highest priority is selected.

We maintain a priority queue (with the key of $rank_u + rank_d$) to contain all *ready* tasks at any given instant. A binary heap was used to implement the priority queue, which has time complexity of $O(log\ v)$ for insertion and deletion of a task and $O(1)$ for retrieving the task with the highest priority. At each step, the task with the highest $rank_u + rank_d$ value is selected from the priority queue.

Processor Selection Phase. The critical-path processor, p_{CP}, is the one that minimizes the cumulative computation costs of the tasks on the critical path (Step 13). If the selected task is on the critical path, then it is scheduled on the critical-path processor; otherwise, it is assigned to a processor which minimizes the earliest execution finish time of the task. Both cases consider an insertion-based scheduling policy. The time-complexity of the CPOP algorithm is equal to $O(e \times p)$. Fig. 4b shows the schedule obtained by the CPOP algorithm for Fig. 3, which has a schedule length of 86. Based on the values in Table 1, the critical path in Fig. 3 is $\{n_1, n_2, n_9, n_{10}\}$. If all critical path tasks are scheduled on $P1$, $P2$, or $P3$, the path length will be 66, 54, or 63, respectively. $P2$ is selected as the critical path processor. The scheduling order of the tasks with respect to CPOP algorithm is $\{n_1, n_2, n_3, n_7, n_4, n_5, n_9, n_6, n_8, n_{10}\}$.

5 Experimental Results and Discussion

In this section, we present the comparative evaluation of our algorithms and the related work given in Section 3.1. For this purpose, we consider two sets of graphs as the workload for testing the algorithms: randomly generated application graphs and the graphs that represent some of the numerical real world problems. First, we present the metrics used for performance evaluation, which is followed by two sections on experimental results.

5.1 Comparison Metrics

The comparisons of the algorithms are based on the following four metrics:

- **Schedule Length Ratio (SLR).** The main performance measure of a scheduling algorithm on a graph is the schedule length (*makespan*) of its output schedule. Since a large set of task graphs with different properties is used, it is necessary to normalize the schedule length to a lower bound,

1. Set the computation costs of tasks and communication costs of edges with mean values.
2. Compute $rank_u$ of tasks by traversing graph upward, starting from the exit task.
3. Compute $rank_d$ of tasks by traversing graph downward, starting from the entry task.
4. Compute $priority(n_i) = rank_d(n_i) + rank_u(n_i)$ for each task n_i in the graph.
5. $|CP| = priority(n_{entry})$, where n_{entry} is the *entry* task.
6. $SET_{CP} = \{n_{entry}\}$, where SET_{CP} is the set of tasks on the critical path.
7. $n_k \leftarrow n_{entry}$.
8. **while** n_k is not the exit task **do**
9. Select n_j where $((n_j \in succ(n_k))$ **and** $(priority(n_j) == |CP|))$.
10. $SET_{CP} = SET_{CP} \bigcup \{n_j\}$.
11. $n_k \leftarrow n_j$.
12. **endwhile**
13. Select the critical-path processor (p_{CP}) which minimizes $\sum_{n_i \in SET_{CP}} w_{i,j}, \quad \forall p_j \in Q$.
14. Initialize the priority queue with the entry task.
15. **while** there is an unscheduled task in the priority queue **do**
16. Select the highest priority task n_i from priority queue.
17. **if** $n_i \in SET_{CP}$ **then**
18. Assign the task n_i on p_{CP}.
19. **else**
20. Assign the task n_i to the processor p_j which minimizes the $EFT(n_i, p_j)$.
21. Update the priority-queue with the successors of n_i, if they become ready tasks.
22. **endwhile**

Fig. 5. The CPOP algorithm.

which is called the Schedule Length Ratio (SLR). The SLR value of an algorithm on a graph is defined by

$$SLR = \frac{makespan}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in Q}\{w_{i,j}\}}. \quad (11)$$

The denominator is the summation of the minimum computation costs of tasks on the CP_{MIN}. (For an unscheduled DAG, if the computation cost of each node n_i is set with the minimum value, then the critical path will be based on minimum computation costs, which is represented as CP_{MIN}.) The SLR of a graph (using any algorithm) cannot be less than one since the denominator is the lower bound. The task-scheduling algorithm that gives the lowest SLR of a graph is the best algorithm with respect to performance. Average SLR values over several task graphs are used in our experiments.

- **Speedup**. The speedup value for a given graph is computed by dividing the sequential execution time (i.e., cumulative computation costs of the tasks in the graph) by the parallel execution time (i.e., the makespan of the output schedule). The sequential execution time is computed by assigning all tasks to a single processor that minimizes the cumulative of the computation costs.

$$Speedup = \frac{\min_{p_j \in Q}\{\sum_{n_i \in V} w_{i,j}\}}{makespan}. \quad (12)$$

If the sum of the computation costs is maximized, it results in a higher speedup, but ends up with the same ranking of the scheduling algorithms. Efficiency, the ratio of the speedup value to the number of processors used, is another comparison metric used for application graphs of real world problems given in Section 5.3.

- **Number of Occurrences of Better Quality of Schedules.** The number of times that each algorithm produced better, worse, and equal quality of schedules compared to every other algorithm is counted in the experiments.
- **Running Time of the Algorithms**. The running time (or the scheduling time) of an algorithm is its execution time for obtaining the output schedule of a given task graph. This metric basically gives the average cost of each algorithm. Among the algorithms that give comparable SLR values, the one with the minimum running time is the most practical implementation. The minimization of SLR by checking all possible task-processor pairs can conflict with the minimization in the running time.

5.2 Randomly Generated Application Graphs

In our study, we first considered the randomly generated application graphs. A random graph generator was implemented to generate weighted application DAGs with various characteristics that depend on several input parameters given below. Our simulation-based framework allows assigning sets of values to the parameters used by random graph generator. This framework first executes the random graph generator program to construct the application DAGs, which is followed by the execution of the scheduling algorithms to generate output schedules, and, finally, it computes the performance metrics based on the schedules.

5.2.1 Random Graph Generator

Our random graph generator requires the following input parameters to build weighted DAGs.

- Number of tasks in the graph, (v).
- Shape parameter of the graph, (α). We assume that the height (depth) of a DAG is randomly generated from a uniform distribution with a mean value equal to $\frac{\sqrt{v}}{\alpha}$. (The height is equal to the smallest integral value not less than the real value generated randomly.) The width for each level is randomly selected from a uniform distribution with mean equal to $\alpha \times \sqrt{v}$. A dense graph (a shorter graph with high parallelism) can be generated by selecting $\alpha >> 1.0$; if $\alpha << 1.0$, it will generate a longer graph with a low parallelism degree.
- Out degree of a node, (out_degree).
- Communication to computation ratio, *(CCR)*. It is the ratio of the average communication cost to the average computation cost. If a DAG's CCR value is very low, it can be considered as a computation-intensive application.
- Range percentage of computation costs on processors, (β). It is basically the heterogeneity factor for processor speeds. A high percentage value causes a significant difference in a task's computation cost among the processors and a low percentage indicates that the expected execution time of a task is almost equal on any given processor in the system. The average computation cost of each task n_i in the graph, i.e., $\overline{w_i}$, is selected randomly from a uniform distribution with range $[0\,,\, 2 \times \overline{w_{DAG}}]$, where $\overline{w_{DAG}}$ is the average computation cost of the given graph, which is set randomly in the algorithm. Then, the computation cost of each task n_i on each processor p_j in the system is randomly set from the following range:

$$\overline{w_i} \times \left(1 - \frac{\beta}{2}\right) \le w_{i,j} \le \overline{w_i} \times \left(1 + \frac{\beta}{2}\right). \quad (13)$$

In each experiment, the values of these parameters are assigned from the corresponding sets given below. A parameter should be assigned by all values given in its set in a single experiment and, in case of any change on these values, it is written explicitly in the paper. Note that the last value in the out_degree set is the number of nodes in the graph which generate fully connected graphs for the experiments.

- $SET_V = \{20, 40, 60, 80, 100\}$,
- $SET_{CCR} = \{0.1, 0.5, 1.0, 5.0, 10.0\}$,
- $SET_\alpha = \{0.5, 1.0, 2.0\}$,
- $SET_{out\_degree} = \{1, 2, 3, 4, 5, v\}$,
- $SET_\beta = \{0.1, 0.25, 0.5, 0.75, 1.0\}$.

These combinations give 2,250 different DAG types. Since 25 random DAGs were generated for each DAG type, the total number of DAGs used in our experiments was around $56K$. Assigning several input parameters and selecting each parameter from a large set cause the

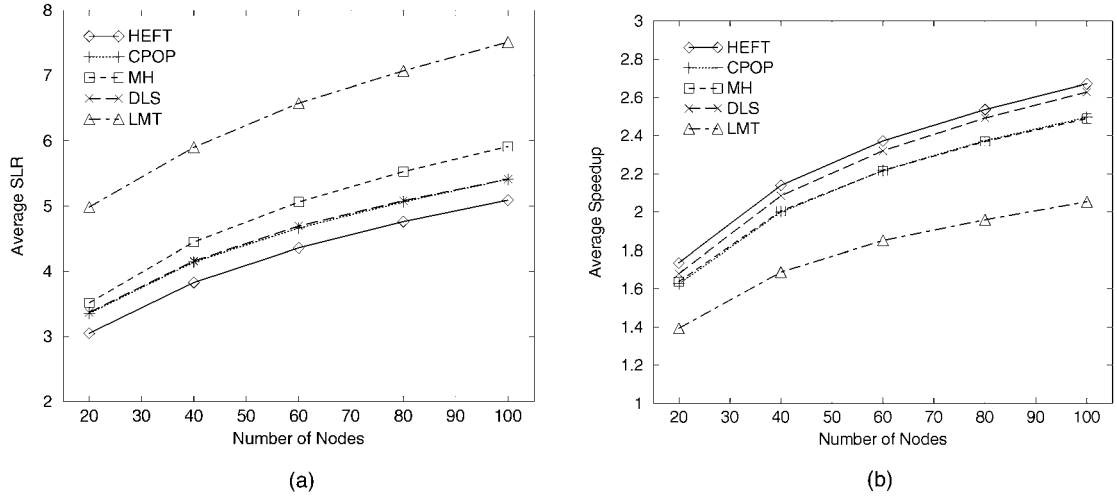


Fig. 6. (a) Average SLR and (b) average speedup with respect to graph size.

generation of diverse DAGs with various characteristics. Experiments based on diverse DAGs prevent biasing toward a particular scheduling algorithm.

5.2.2 Performance Results

The performance of the algorithms were compared with respect to various graph characteristics. The first set of experiments compares the performance and cost of the algorithms with respect to various graph sizes (see Figs. 6 and 7). The SLR-based performance ranking of the algorithms is {HEFT, CPOP, DLS, MH, LMT}. (It should be noted that each ranking in this paper starts with the best algorithm and ends with the worst one with respect to the given comparison metric.) The average SLR value of HEFT on all generated graphs is better than the CPOP algorithm by 7 percent, the DLS algorithm by 8 percent, the MH algorithm by 16 percent, and the LMT algorithm by 52 percent. The average speedup ranking of the algorithms is {HEFT, DLS, (CPOP=MH), LMT} (Fig. 6b).

Based on these experiments, the HEFT algorithm outperforms the other algorithms for any graph size in terms of SLR and speedup. The CPOP algorithm outperforms the related work in terms of average SLR; for various graph sizes, it cannot give higher speedup values than the DLS algorithm. With respect to average running times (see Fig. 7), The HEFT algorithm is the fastest and the DLS algorithm is the slowest one. On average, the HEFT algorithm is faster than the CPOP algorithm by 10 percent, the MH algorithm by 32 percent, the DLS algorithm by 84 percent, and the LMT algorithm by 48 percent.

The next experiment is with respect to the graph structure. When α (the shape parameter of the graph) is equal to 0.5, i.e., the generated graphs have greater depths with a low degree of parallelism, it is shown that the performance of the HEFT algorithm is better than that of the CPOP algorithm by 8 percent, the MH algorithm by 12 percent, the DLS algorithm by 16 percent, and the LMT algorithm by 40 percent. When α is equal to 1.0, the average SLR value of the HEFT algorithm is better than that of the CPOP algorithm by 7 percent, the MH algorithm by 14 percent, the DLS algorithm by 7 percent, and the LMT algorithm by 34 percent. When α is equal to 2.0, the HEFT algorithm is better than the CPOP algorithm by 6 percent, the MH algorithm by 15 percent, the DLS algorithm by 8 percent, and the LMT algorithm by 31 percent. For all three different graph structures, the HEFT algorithm gives the best performance.

Quality of schedules generated by the algorithms with respect to various CCR values was compared in another experiment. The performance ranking of the algorithms when $CCR \leq 1.0$ is {HEFT, DLS, MH, CPOP, LMT}. When $CCR > 1.0$, the performance ranking changes to {HEFT, CPOP, DLS, MH, LMT}. The CPOP algorithm gives better results for graphs with higher CCRs than the graphs with lower CCRs. Clustering of the critical path on the fastest processor results in better quality of schedules for the graphs in which average communication cost is greater than average computation cost.

Finally, the number of times that each scheduling algorithm in the experiments produced better, worse, or equal schedule length compared to every other algorithm was counted for the 56250 DAGs used. Each cell in Table 2 indicates the comparison results of the algorithm on the left with the algorithm on the top. The "combined" column shows the percentage of graphs in which the algorithm on the left gives a better, equal, or worse performance than all other algorithms combined. The ranking of the algorithms,

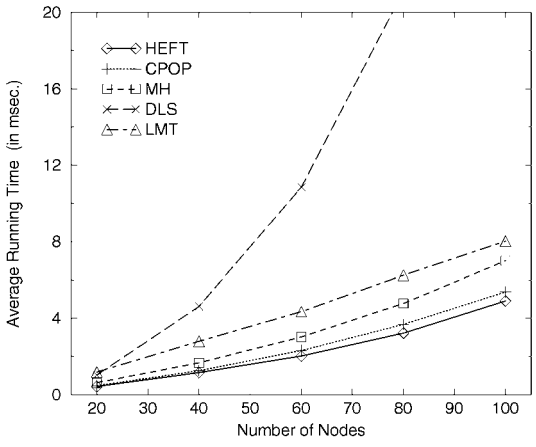


Fig. 7. Average running time of algorithms with respect to graph size.

TABLE 2
Pair-Wise Comparison of the Scheduling Algorithms

		HEFT	CPOP	DLS	MH	LMT	Combined
HEFT	better		45181	42709	49730	56059	86%
	equal	*	215	802	689	2	1%
	worse		10854	12739	5831	189	13%
CPOP	better	10854		24774	34689	53922	55%
	equal	215	*	108	76	3	< 1%
	worse	45181		31368	21485	2325	45%
DLS	better	12739	31368		44056	55873	64%
	equal	802	108	*	2170	1	1%
	worse	42709	24774		10024	376	35%
MH	better	5831	21485	10024		55342	41%
	equal	689	76	2170	*	6	1%
	worse	49730	34689	44056		902	58%
LMT	better	189	2325	376	902		2%
	equal	2	3	1	6	*	< 1%
	worse	56059	53922	55873	55342		98%

based on occurrences of best results, is {*HEFT, DLS, CPOP, MH, LMT*}. However, the ranking with respect to average SLR values was: {*HEFT, CPOP, DLS, MH, LMT*}. Although the DLS algorithm outperforms the CPOP algorithm in terms of the number of occurrences of best results, the CPOP algorithm has shown slightly better average SLR value than the DLS algorithm.

5.3 Application Graphs of Real World Problems

In addition to randomly generated task graphs, we also considered application graphs of three real world problems: Gauss elimination algorithm [3], [28], Fast Fourier Transformation [29], [30], and a molecular dynamics code given in [19].

5.3.1 Gaussian Elimination

Fig. 8a gives the sequential program for the Gaussian elimination algorithm [3], [28]. The data-flow graph of the algorithm for the special case of $m = 5$, where m is the dimension of the matrix, is given in Fig. 8b. Each $T_{k,k}$ represents a pivot column operation and each $T_{k,j}$ represents an update operation. In Fig. 8b, the critical path is $T_{1,1}T_{1,2}T_{2,2}T_{2,3}T_{3,3}T_{3,4}T_{4,4}T_{4,5}$, which is the path with the maximum number of tasks.

For the experiments of Gauss elimination application, the same CCR and range percentage values (given in Section 5.2) were used. Since the structure of the application graph is known, we do not need the other parameters, such as the number of tasks, out_degree, and shape parameters. A new parameter, matrix size (m), is used in place of v (the number of tasks in the graph). The total number of tasks in a Gaussian elimination graph is equal to $\frac{m^2+m-2}{2}$.

Fig. 9a gives the average SLR values of the algorithms at various matrix sizes from 5 to 20, with an increment of one, when the number of processors is equal to five. The smallest size graph in this experiment has 14 tasks and the largest one has 209 tasks. The performances of the HEFT and DLS algorithms are the best of all. Increasing the matrix size causes more tasks not to be on the critical path, which results in an increase in the makespan for each algorithm.

For the efficiency comparison, the number of processors used in our experiments is varied from 2 to 16, incrementing by the power of 2; the CCR and range percentage parameters have the same set of values. Fig. 9b gives efficiency comparison for Gaussian elimination graphs when the matrix size is 50. The HEFT and DLS algorithms have better efficiency than the other algorithms. When the number of processors is increased beyond eight, the HEFT algorithm outperforms the DLS algorithm in terms of efficiency. Since the matrix size is fixed, an increase in the number of processors decreases the makespan for each algorithm. As part of this experiment, we compared the running time of the algorithms with respect to the various numbers of processors (by keeping the matrix size fixed). The results indicate that the DLS algorithm is the slowest algorithm among them, although it performs as well as the HEFT algorithm. As an example, when the matrix size is 50 for 16 processors, the DLS algorithm takes 16.2 times longer than the HEFT algorithm to schedule a given graph. When the performance and cost results are considered together, the HEFT algorithm is the most efficient and practical algorithm among them.

5.3.2 Fast Fourier Transformation

The recursive, one-dimensional FFT Algorithm [29], [30] and its task graph (when there are four data points) is given in Fig. 10. In this figure, A is an array of size m which holds the coefficients of the polynomial and array Y is the output of the algorithm. The algorithm consists of two parts: recursive calls (lines 3-4) and the butterfly operation (lines 6-7). The task graph in Fig. 10b can be divided into two parts—the tasks above the dashed line are the recursive call tasks and the ones below the line are butterfly operation tasks. For an input vector of size m, there are $2 \times m - 1$ recursive call tasks and $m \times \log_2 m$ butterfly operation tasks. (We assume that $m = 2^k$ for some integer k). Each path from the start task to any of the exit tasks in an FFT task graph is a critical path since the computation costs of tasks in any level are equal and the communication costs of all edges between two consecutive levels are equal.

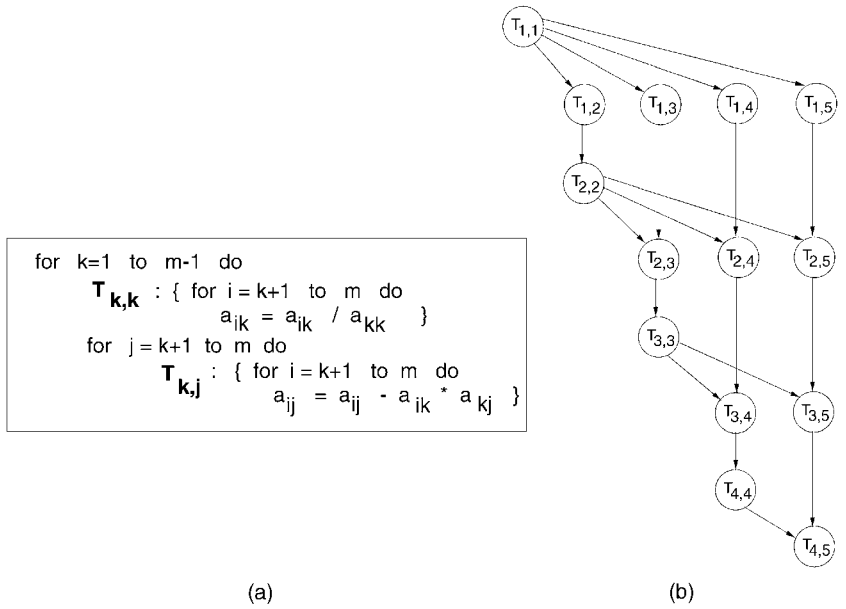


Fig. 8. (a) Gaussian elimination algorithm, (b) task graph for matrix of size 5.

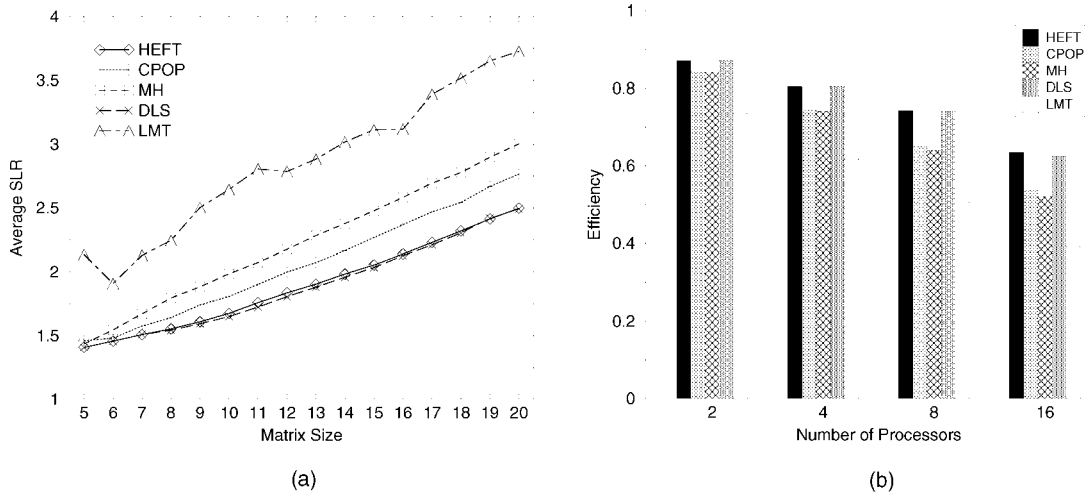


Fig. 9. (a) Average SLR and (b) efficiency comparison for the Gaussian elimination graph.

For the FFT-related experiments, only the CCR and range percentage parameters, among the parameters given in Section 5.2, were used, as in the Gauss elimination application. The number of data points in FFT is another parameter in our simulations, which varies from 2 to 32 incrementing by the power of 2. Fig. 11a shows the average SLR values for FFT graphs at various sizes of input points. One can observe that the HEFT algorithm outperforms the other algorithms in most of the cases. Fig. 11b presents the efficiency values obtained for each of the algorithms with respect to various numbers of processors with graphs of 64 data points. The number of processors used varied from two to 32, incrementing by the power of 2. The HEFT and DLS algorithms give the most efficient schedules in all cases.

When the running times of the algorithms for scheduling FFT graphs are compared with respect to both the number of data points and the number of processors used (see Fig. 12), one can observe that the DLS algorithm is the highest cost algorithm. Note that the number of processors is equal to six in Fig. 12a and the number of input points is equal to 64 in Fig. 12b.

5.3.3 Molecular Dynamics Code

Fig. 13 is the task graph of a modified molecular dynamic code given in [19]. This application is part of our performance evaluation since it has an irregular task graph. Since the number of tasks is fixed in the application and the structure of the graph is known, only the values of CCR and range percentage parameters (in Section 5.2) are used in our experiments. Fig. 14a shows the performance of the algorithms with respect to five different CCR values when the number of processors is equal to six. On the average, the SLR ranking is {HEFT, DLS, CPOP, MH, LMT}. The efficiency comparison of the scheduling algorithms is given in Fig. 14b, in which the number of processors is varied

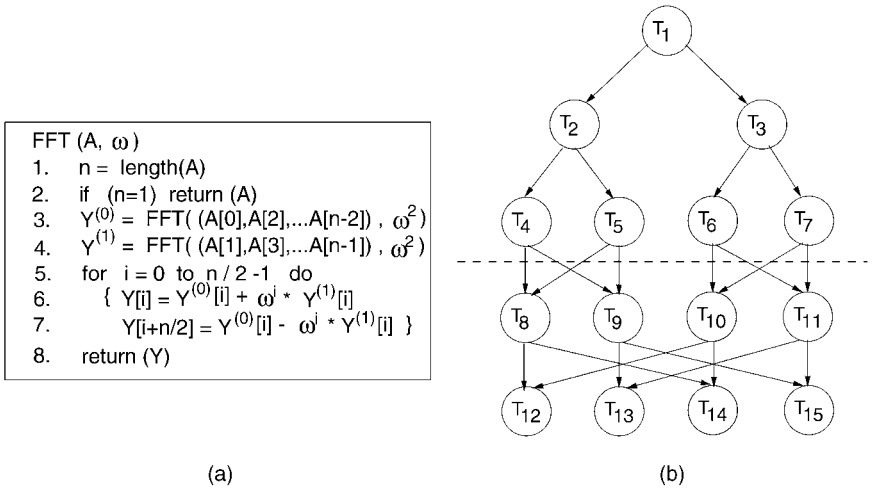


Fig. 10. (a) FFT algorithm, (b) the generated DAG of FFT with four points.

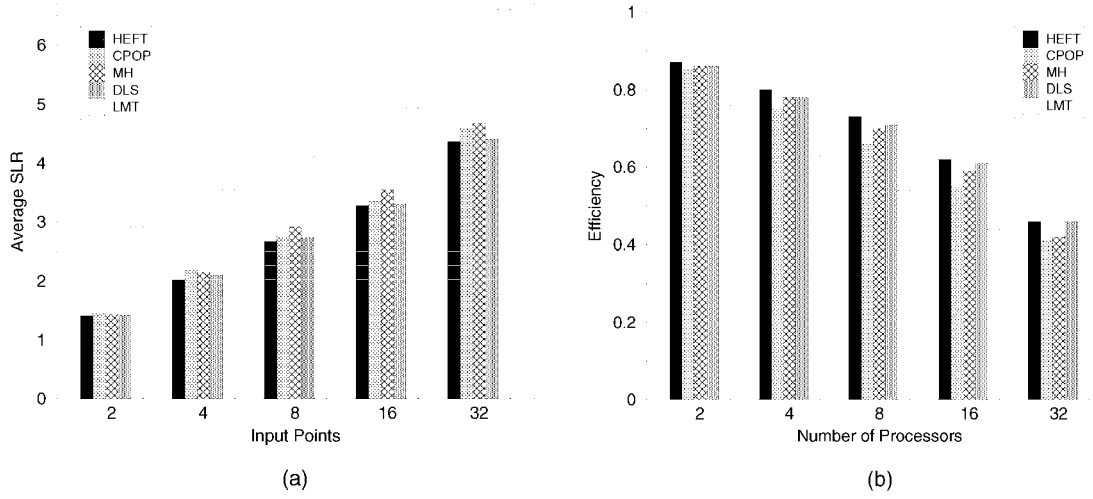


Fig. 11. (a) Average SLR and (b) efficiency comparison for the FFT graph.

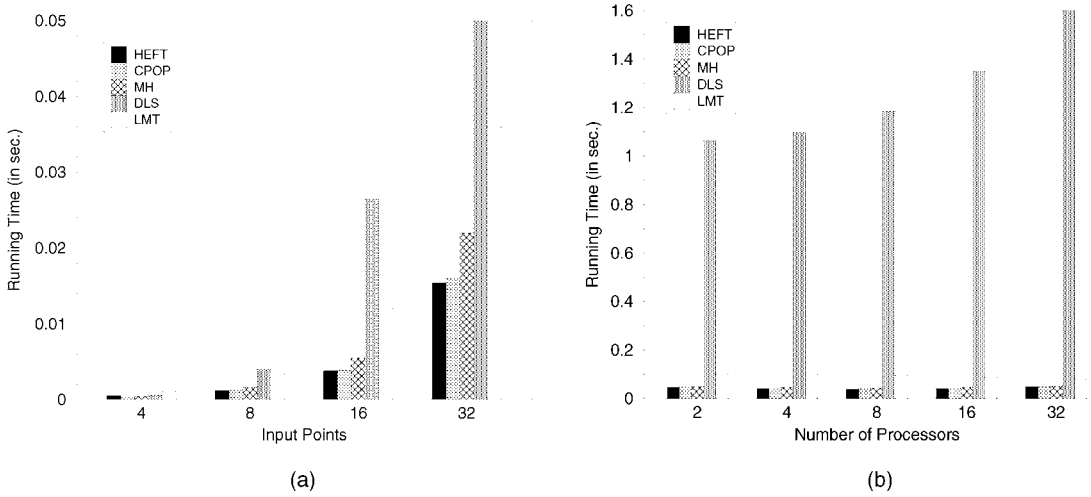


Fig. 12. Running times of scheduling algorithms for the FFT graph.

from two to seven with an increment of 1. Since there are at most seven tasks in any level in Fig. 13, the number of processors in the experiments is bounded up to seven processors. It was also observed that the DLS and LMT algorithms take a running time almost three times longer than the other three algorithms (HEFT, CPOP, and MH). When these results are combined, the HEFT algorithm is the most practical and efficient algorithm for this application.

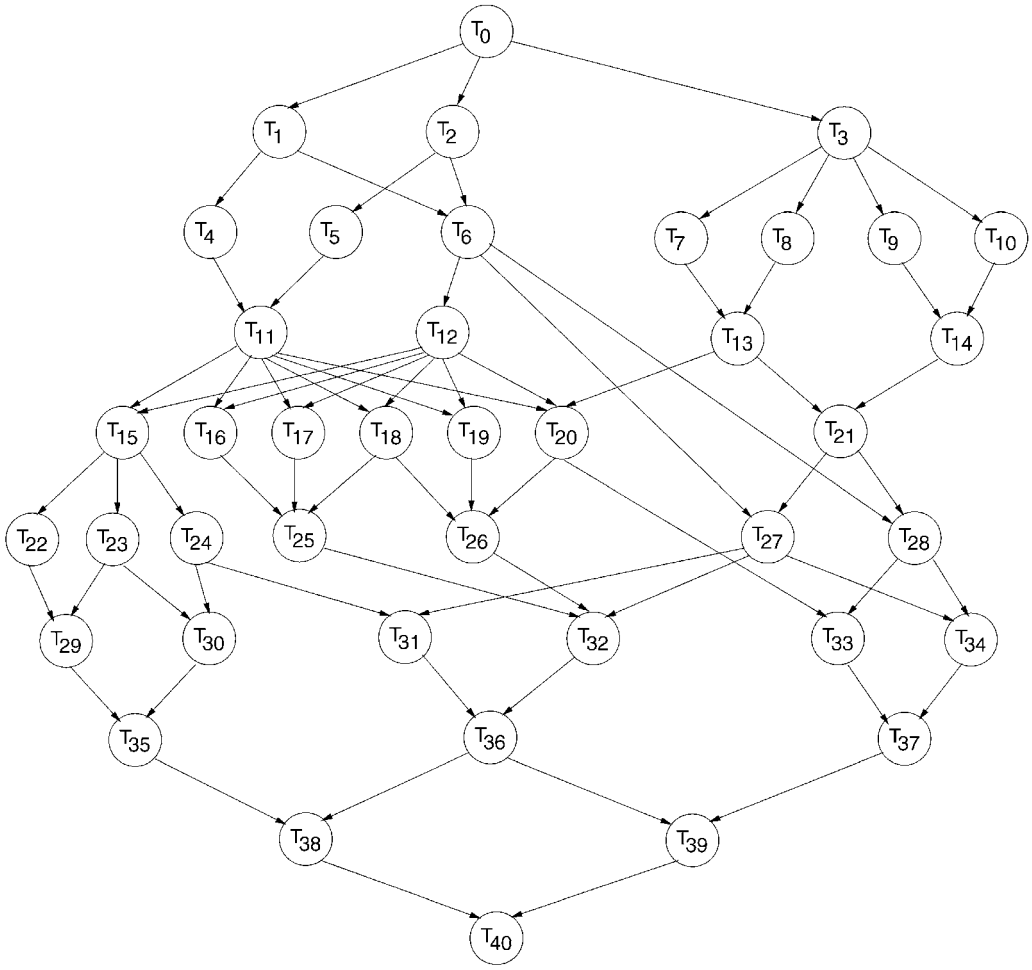


Fig. 13. The task graph of the molecular dynamics code [19].

6 Alternate Policies for the Phases of the HEFT Algorithm

We proposed three substitute policies (shown as A1, A2, and A3) for the task prioritizing phase of the HEFT algorithm, which is based on upward rank in the experiments. In A1, the priority value is equal to the summation of the upward and downward ranks. In A2, the right part of the $+$ sign gives the latest execution finish time of an immediate predecessor task of n_i, which is already scheduled. A3 is similar to A2, except that it considers the communication cost. In the experiments, it has been observed that the original priority policy gives better results than these alternates by 6 percent.

- **A1**: $priority(n_i) = rank_u(n_i) + rank_d(n_i)$,
- **A2**: $priority(n_i) = rank_u(n_i) + \max_{n_j \in pred(n_i)} AFT(n_j)$,
- **A3**:

$$priority(n_i) = rank_u(n_i) + \max_{n_j \in pred(n_i)} \{AFT(n_j) + c_{j,i}\}.$$

Another extension that may improve performance is to take immediate child tasks into account. As an example, the HEFT algorithm generates an output schedule of length 8 for the task graph given in Fig. 15; if task A and its immediate child (task B) are scheduled on the same processor, which minimizes the earliest finish time of task B, the schedule length decreases to 7. To consider this extension, we can modify the processor selection phase of the HEFT algorithm as follows: For each selected task, one of its immediate child tasks is marked as the critical-child based on one of the three policies given below. If the other immediate predecessors of the critical child are already scheduled, then the selected task and its critical child are scheduled on the same processor that minimizes the earliest finish time of the critical child; otherwise, the selected task is scheduled to the processor that minimizes its earliest finish time, as in the HEFT algorithm. The three critical child selection policies (B1, B2, and B3) use either communication cost or upward rank or both.

- **B1**: $critical\_child(n_i) = \max_{n_c \in succ(n_i)} c_{i,c}$,
- **B2**: $critical\_child(n_i) = \max_{n_c \in succ(n_i)} rank_u(n_c)$,
- **B3**:

$$critical\_child(n_i) = \max_{n_c \in succ(n_i)} \{rank_u(n_c) + c_{i,c}\}.$$

The original HEFT algorithm outperforms these alternates for small CCR graphs. For high CCR graphs, some benefit has been observed by taking critical child tasks into account during processor selection. When $3.0 \leq CCR < 6.0$, B1 policy slightly outperforms the original HEFT algorithm. If $CCR \geq 6.0$, B2 policy outperforms the original algorithm and others alternates by 4 percent.

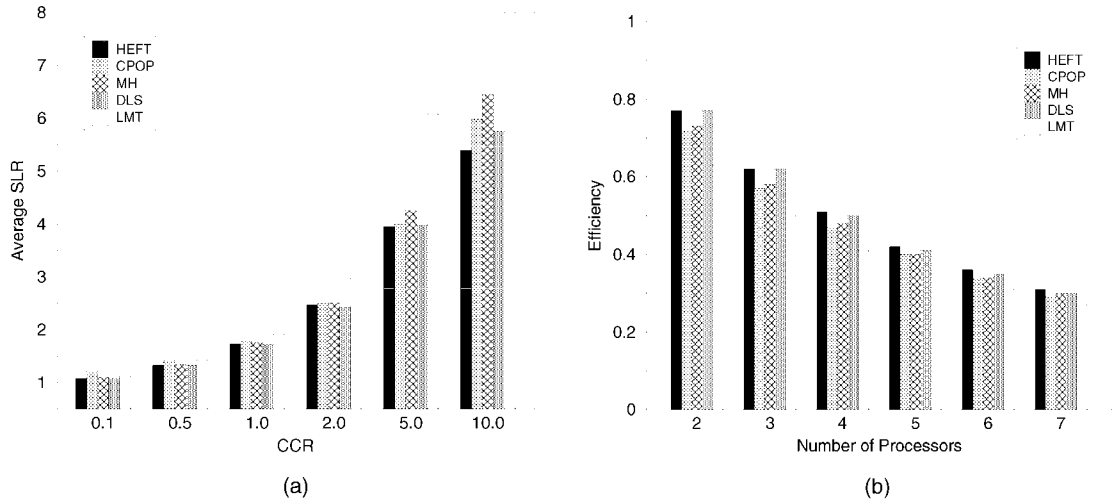


Fig. 14. (a) Average SLR and (b) efficiency comparison for the task graph of a molecular dynamics code.

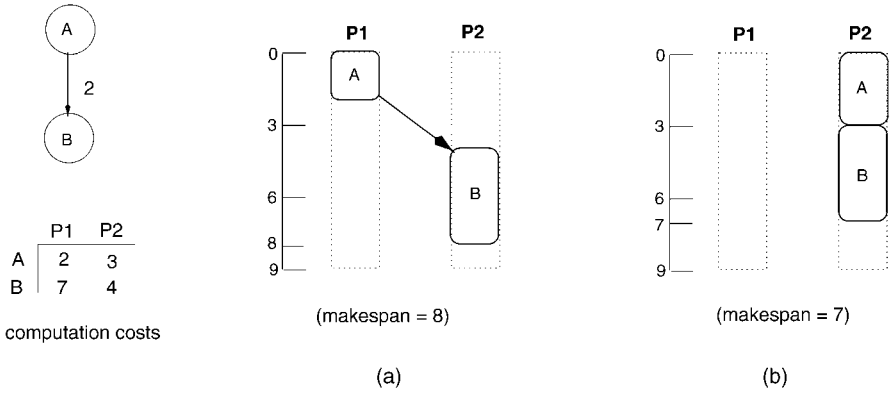


Fig. 15. Scheduling of a task graph with the (a) HEFT algorithm and (b) an alternative method.

7 Conclusions

In this paper, we presented two new algorithms, called the HEFT algorithm and the CPOP algorithm, for scheduling application graphs onto a system of heterogeneous processors. Based on the experimental study using a large set (56K) of randomly generated application graphs with various characteristics and application graphs of several real world problems (such as Gaussian elimination, FFT, and a molecular dynamics code), the HEFT algorithm significantly outperformed the other algorithms in terms of both performance and cost metrics, including average schedule length ratio, speedup, frequency of best results, and average running time. Because of its robust performance, low running time, and the ability to give stable performance over a wide range of graph structures, the HEFT algorithm is a viable solution for the DAG scheduling problem on heterogeneous systems. Based on our performance evaluation study, we also observed that the CPOP algorithm has given either better performance and better running time results than existing algorithms or comparable results with them.

Several alternative policies were studied for task prioritizing and processor selection phases of the HEFT algorithm. A new method was introduced for the processor selection phase which tries to minimize the earliest finish time of the critical-child task of each selected task.

One planned future research is to analytically investigate the trade-off between the quality of schedules of the algorithms, i.e., average makespan values, and the number of processors available. This extension may come up with some bounds on the degradation of makespan given that the number of processors available may not be sufficient. We plan to extend the HEFT Algorithm for rescheduling tasks in response to changes in processor and network loads. Although our algorithms assume a fully connected network, it is also planned to extent these algorithms for arbitrary-connected networks by considering the link contention.

References

[1] M.R. Gary and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., 1979.
[2] J.D. Ullman, "NP-Complete Scheduling Problems," *J. Computer and Systems Sciences,* vol. 10, pp. 384-393, 1975.
[3] M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message Passing Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, pp. 330-343, July 1990.
[4] Y. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 5, pp. 506-521, May 1996.

[5] E.S.H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Trans. Parallel and Distributed Systems* vol. 5, no. 2, pp. 113-120, Feb. 1994.
[6] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 2, pp. 175-186, Feb. 1993.
[7] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *J. Parallel and Distributed Computing,* vol. 9, pp. 138-153, 1990.
[8] H. Singh and A. Youssef, "Mapping and Scheduling Heterogeneous Task Graphs Using Genetic Algorithms," *Proc. Heterogeneous Computing Workshop,* pp. 86-97, 1996.
[9] I. Ahmad and Y. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," *Proc. Int'l Conf. Parallel Processing,* vol. 2, pp. 47-51, 1994.
[10] M. Iverson, F. Ozguner, and G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environment," *Proc. Heterogeneous Computing Workshop,* pp. 93-100, 1995.
[11] P. Shroff, D.W. Watson, N.S. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," *Proc. Heterogeneous Computing Workshop,* pp. 98-104, 1996.
[12] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems,* vol. 5, no. 9, pp. 951-967, Sept. 1994.
[13] L. Wang, H.J. Siegel, and V.P. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments," *Proc. Heterogeneous Computing Workshop,* 1996.
[14] M. Maheswaran and H.J. Siegel, "A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems," *Proc. Heterogeneous Computing Workshop,* pp. 57-69, 1998.
[15] L. Tao, B. Narahari, and Y.C. Zhao, "Heuristics for Mapping Parallel Computations to Heterogeneous Parallel Architectures," *Proc. Heterogeneous Computing Workshop,* 1993.
[16] M. Wu, W. Shu, and J. Gu, "Local Search for DAG Scheduling and Task Assignment," *Proc. 1997 Int'l Conf. Parallel Processing,* pp. 174-180, 1997.
[17] R.C. Correa, A. Ferreria, and P. Rebreyend, "Integrating List Heuristics into Genetic Algorithms for Multiprocessor Scheduling," *Proc. Eighth IEEE Symp. Parallel and Distributed Processing (SPDP '96),* Oct. 1996.
[18] B. Kruatrachue and T.G. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software,* pp. 23-32, Jan. 1988.
[19] S.J. Kim and J.C. Browne, "A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures," *Proc. Int'l Conf. Parallel Processing,* vol. 2, pp. 1-8, 1988.
[20] Y. Kwok, I. Ahmad, and J. Gu, "FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors," *Proc. Int'l Conf. Parallel Processing,* vol. 2, pp. 150-157, 1996.
[21] Y. Kwok and I. Ahmad, "Benchmarking the Task Graph Scheduling Algorithms," *Proc. First Merged Int'l Parallel Processing Symp./Symp. Parallel and Distributed Processing Conf.,* pp. 531-537, 1998.
[22] J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Costs," *SIAM J. Computing,* vol. 18, no. 2, pp. 244257, 1989.
[23] H. El-Rewini, H.H. Ali, and T. Lewis, "Task Scheduling in Multiprocessor Systems," *Computer,* pp. 27-37, Dec. 1995.
[24] J. Liou and M.A. Palis, "A Comparison of General Approaches to Multiprocessor Scheduling," *Proc. Int'l Parallel Processing Symp.,* pp. 152-156, 1997.
[25] J. Liou and M.A. Palis, "An Efficient Clustering Heuristic for Scheduling DAGs on Multiprocessors," *Proc. Symp. Parallel and Distributed Processing,* 1996.
[26] A. Radulescu, A.J.C. van Gemund, and H. Lin, "LLB: A Fast and Effective Scheduling Algorithm for Distributed-Memory Systems," *Proc. Second Merged Int'l Parallel Processing Symp./Symp. Parallel and Distributed Processing Conf.,* 1999.
[27] G. Park, B. Shirazi, and J. Marquis, "DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems," *Proc. Int'l Conf. Parallel Processing,* pp. 157-166, 1997.
[28] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, "Parallel Gaussian Elimination on an MIMD Computer," *Parallel Computing,* vol. 6, pp. 275-295, 1988.
[29] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press, 1990.
[30] Y. Chung and S. Ranka, "Applications and Performance Analysis of a Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors," *Proc. Supercomputing,* pp. 512-521, Nov. 1992.
[31] T. Braun, H.J. Siegel, N. Beck, L.L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hengsen, and R.F. Freund, "A Comparison Study of Static Mapping Heuristics for a Class of Meta-Tasks on Heterogeneous Computing Systems," *Proc. Heterogeneous Computing Workshop,* pp. 15-29, 1999.

Haluk Topcuoglu received the BS and MS degrees in computer engineering from Bogazici University, Istanbul, Turkey, in 1991 and 1993, respectively. He received the PhD degree in computer science from Syracuse University, New York, in 1999. He is currently an assistant professor in the Computer Engineering Department, Marmara University, Turkey. His research interests include task scheduling techniques in heterogeneous environments, cluster computing, parallel and distributed programming, web technologies, and genetic algorithms. He is a member of the IEEE, the IEEE Computer Society and the ACM.

Salim Hariri received the MSc degree from Ohio State University in 1982 and the PhD degree in computer engineering from the University of Southern California in 1986. He is a professor in the Electrical and Computer Engineering Department at the University of Arizona and the director of the Center for Advanced TeleSysMatics (CAT): Next-Generation Network-Centric Systems. He is the editor in chief for *Cluster Computing: The Journal of Networks, Software Tools, and Applications*. His current research focuses on high performance distributed computing, agent-based proactive and intelligent network management systems, design and analysis of high speed networks, and developing software design tools for high performance computing and communication systems and applications. He has coauthored more than 200 journal and conference research papers and is the author of the book, *High Performance Distributed Computing: Network, Architecture and Programming*, to be published by Prentice Hall, in 2002. He is a member of the IEEE Computer Society.

Min-You Wu received the MS degree from the Graduate School of Academia Sinica, Beijing, China, and the PhD degree from Santa Clara University, California. He is an associate professor in the Department of Electrical and Computer Engineering at the University of New Mexico. He has held various positions at the University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, the State University of New York at Buffalo, and the University of Central Florida. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published more than 90 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a senior member of the IEEE and a member of ACM. He is listed in *International Who's Who of Information Technology and Who's Who in America*.