Task Scheduling Algorithms for Heterogeneous Processors

Haluk Topcuoglu
Department of Electrical Engineering and Computer Science
Syracuse University, Syracuse, NY 13244-4100
haluk@top.cis.syr.edu

Salim Hariri
Department of Electrical and Computer Engineering
The University of Arizona, Tucson, Arizona 85721-0104

Min-You Wu
Department of Electrical and Computer Engineering
University of Central Florida

Abstract

Scheduling computation tasks on processors is the key issue for high-performance computing. Although a large number of scheduling heuristics have been presented in the literature, most of them target only homogeneous resources. The existing algorithms for heterogeneous domains are not generally efficient because of their high complexity and/or the quality of the results. We present two low-complexity efficient heuristics, the Heterogeneous Earliest-Finish-Time (HEFT) Algorithm and the Critical-Path-on-a-Processor (CPOP) Algorithm for scheduling directed acyclic weighted task graphs (DAGs) on a bounded number of heterogeneous processors. We compared the performances of these algorithms against three previously proposed heuristics. The comparison study showed that our algorithms outperform previous approaches in terms of performance (schedule length ratio and speedup) and cost (time-complexity).

1. Introduction

Efficient scheduling of application tasks is critical to achieving high performance in parallel and distributed systems. The objective of scheduling is to map the tasks onto the processors and order their execution so that task precedence requirements are satisfied and minimum schedule length (or *makespan*) is given. Since the general DAG scheduling is NP-complete, there are many research efforts that have proposed heuristics for the task scheduling problem [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13].

Although a wide variety of different approaches are used to solve the DAG scheduling problem, most of them target only for homogeneous processors. The scheduling techniques that are suitable for homogeneous domains are limited and may not be suitable for heterogeneous domains. Only a few methods [5, 4, 9, 10] use variable execution times of tasks for heterogeneous environments; however, they are either high-complexity algorithms and/or they do not generally provide good quality of results.

In this paper we propose two static DAG scheduling algorithms for heterogeneous environments. They are for a bounded number of processors and are based on list-scheduling heuristics. The Heterogeneous Earliest-Finish-Time (HEFT) Algorithm selects the task with the highest upward rank (defined in Section 2) at each step; then the task is assigned to the most suitable processor that minimizes the earliest finish time with an insertion-based approach. The Critical-Path-on-a-Processor (CPOP) Algorithm schedules critical-path nodes onto a single processor that minimizes the critical path length. For the other nodes, the task selection phase of the algorithm is based on a summation of downward and upward ranks; the processor selection phase is based on the earliest execution finish time, as in the HEFT Algorithm. The simulation study in Section 5 shows that our algorithms considerably outperform previous approaches in terms of performance

(schedule length ratio and speed-up) and cost (time-complexity).

The remainder of this paper is organized as follows. The next section gives the background of the scheduling problem, including some definitions and parameters used in the algorithms. In Section 3 we present the proposed scheduling algorithms for heterogeneous domains. Section 4 contains a brief review on the related scheduling algorithms that will be used in our comparison, and in Section 5 the performances of our algorithms are compared with the performances of related work, using task graphs of some real applications and randomly generated tasks graphs. Section 6 includes the conclusion and future work.

2. Problem Definition

A parallel/distributed application is decomposed into multiple tasks with data dependencies among them. In our model an application is represented by a directed acyclic graph (DAG) that consists of a tuple $G = (V, E, P, W, data, rate)$, where V is the set of v nodes/tasks, E is the set of e edges between the nodes, and P is the set of processors available in the system. (In this paper *task* and *node* terms are used interchangeably used.) Each edge $(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started.

W is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. Each $w_{i,j}$ gives the estimated execution time to complete task n_i on processor p_j. The average execution costs of tasks are used in the task priority equations. The average execution cost of a node n_i is defined as $\overline{w_i} = \sum_j^p w_{i,j}/p$. $Data$ is a $v \times v$ matrix for data transfer size (in bytes) between the tasks. The data transfer rates (in bytes/second) between processors are stored in a $p \times p$ matrix, $rate$.

The communication cost of the edge (i, j), which is for data transfer from task n_i (scheduled on p_m) to task n_j (scheduled on p_n), is defined by $c_{i,j} = data(n_i, n_j)/rate(p_m, p_n)$. When both n_i and n_j are scheduled on the same processor, $p_m = p_n$, then $c_{i,j}$ becomes zero, since the intra-processor communication cost is negligible compared with the interprocessor communication cost. The average communication cost of an edge is defined by $\overline{c_{i,j}} = data(n_i, n_j)/\overline{rate}$, where $\overline{rate}$ is the average transfer rate between the processors in the domain.

The $EST(n_i, p_j)$ and $EFT(n_i, p_j)$ are the *earliest* execution start time and the *earliest* execution finish time of node n_i on processor p_j, respectively. They are defined by

$$EST(n_i, p_j) = \max\{\, T\_Available[j], \max_{n_m \in pred(n_i)} (EFT(n_m, p_k) + c_{m,i}) \,\} \quad (1)$$

$$EFT(n_i, p_j) = w_{i,j} + EST(n_i, p_j) \quad (2)$$

where $pred(n_i)$ is the set of immediate predecessors of task n_i, and $T\_Available[j]$ is the earliest time at which processor p_j is available for task execution. 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 the host p_j. The assignment decisions are stored in a two-dimensional matrix *list*. The j'th row of *list* matrix is for the sequence of nodes (in the order of execution start time) that was already scheduled on p_j.

The *objective function* of the scheduling problem is to determine an assignment of tasks of a given application to processors such that the *schedule length* (or *makespan*) is minimized by satisfying all precedence constraints. After all nodes in the DAG are scheduled, the schedule length will be the earliest finish time of the exit node n_e, $EFT(n_e, p_j)$, where exit node n_e is scheduled to processor p_j. (If a graph has multiple exit nodes, they are connected with zero-weight edges to a *pseudo exit node* that has zero computation cost. Similarly, a pseudo start node is added to the graphs with multiple start nodes).

The *critical path* (CP) of a DAG is the longest path from the entry node to the exit node in the graph. The length of this path, $|CP|$, is the sum of the computation costs of the nodes and inter-node communication costs along the path. The $|CP|$ value of a DAG is the lower bound of the schedule length.

In our algorithms we rank tasks as upward and downward to set the scheduling priorities. The *upward rank* of a task n_i is recursively defined by

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} (\overline{c_{i,j}} + rank_u(n_j)) \quad (3)$$

where $succ(n_i)$ is the set of immediate successors of task n_i. Since it is computed recursively by traversing the task graph upward, starting from the exit node, it is referred to as an *upward* rank. Basically, $rank_u(n_i)$ is the length of the critical path (i.e., the longest path)

from n_i to the exit node, including the computation cost of the node itself. In some previous algorithms the ranks of the nodes are computed using computation costs only, which is referred to as *static* upward rank, $rank_u^s$.

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

$$rank_d(n_i) = \max_{n_j \in pred(n_i)} \{rank_d(n_j) + \overline{w_j} + \overline{c_{j,i}}\} \quad (4)$$

The downward ranks are computed recursively by traversing the task graph downward Basically, the $rank_d(n_i)$ is the longest distance from the start node to the node n_i, excluding the computation cost of the node itself.

In some previous algorithms the *level* attribute is used to set the priorities of the tasks. The level of a task is computed by the maximum number of edges along any path to the task from the start node. The start node has a level of zero.

3. Proposed Algorithms

In this section, we present our scheduling algorithms the Heterogeneous Earliest Finish Time (HEFT) Algorithm and the Critical-Path-on-a-Processor (CPOP) Algorithm.

3.1. The HEFT Algorithm

The Heterogeneous-Earliest-Finish-Time (HEFT) Algorithm, as shown in Figure 1, is a DAG scheduling algorithm that supports a bounded number of heterogeneous processing elements (PEs). To set priority to a task n_i, the HEFT algorithm uses the upward rank value of the task, $rank_u$ (Equation 3), which is the length of the longest path from n_i to the exit node. The $rank_u$ calculation is based on mean computation and communication costs. The task list is generated by sorting the nodes with respect to the decreasing order of the $rank_u$ values. In our implementation the ties are broken randomly; i.e., if two nodes to be scheduled have equal $rank_u$ values, one of them is selected randomly.

The HEFT algorithm uses the earliest finish time value, EFT, to select the processor for each task. In noninsertion-based scheduling algorithms, the earliest available time of a processor p_j, the $T\_Available[j]$ term in Equation 1, is the execution completion time of the last assigned node on p_j. The HEFT Algorithm, which is insertion-based, considers a possible insertion of each task in an earliest idle time slot between two already-scheduled tasks on the given processor. Formally, node n_i can be scheduled on processor p_j, which holds the following in equality for the minimum value of k

$$EST(list_{j,k+1}, p_j) - EFT(list_{j,k}, p_j) \geq w_{i,j} \quad (5)$$

where the $list_{j,k}$ is the kth node (in the start time sequence) that was already assigned on the processor p_j. Then, $T\_Available[j]$ will be equal to $EFT(list_{j,k}, p_j)$. The time complexity of the HEFT Algorithm is equal to $O(v^2 \times p)$.

3.2. The CPOP Algorithm

The Critical-Path-on-a-Processor (CPOP) Algorithm, shown in Figure 2, is another heuristic for scheduling tasks on a bounded number of heterogeneous processors. The $rank_u$ and $rank_d$ attributes of nodes are computed using mean computation and communication costs. The critical path nodes (CPN_i) are determined at Steps 5-6. The critical-path-processor (CPP) is the one that minimizes the length of the critical path (Step 7). The CPOP Algorithm uses $rank_d(n_i) + rank_u(n_i)$ to assign the node priority. The processor selection phase has two options: If the current node is on the critical path, it is assigned to the critical path processor (CPP); otherwise, it is assigned to the processor that minimizes the execution completion time. The latter option is insertion-based (as in the HEFT algorithm). At each iteration we maintain a priority queue to contain all free nodes and select the node that maximizes $rank_d(n_i) + rank_u(n_i)$. A binary heap was used to implement the priority queue, which has time complexity of $O(logv)$ for insertion and deletion of a node and $O(1)$ for retrieving the node with the highest priority (the root node of the heap). The time complexity of the algorithm is $O(v^2 \times p)$ for v nodes and p processors.

4. Related Work

Only a few of the proposed task scheduling heuristics support variable computation and communication costs for heterogeneous domains: the Dynamic Level Scheduling(DLS) Algorithm [4], the Levelized-Min Time (LMT) Algorithm [9], and the Mapping Heuristic (MH) Algorithm [5]. Although there are genetic algorithm based research efforts [6, 10, 14], most of them are slow and usually do not perform as well as the list-scheduling algorithms.

1. Compute $rank_u$ for all nodes by traversing graph upward, starting from the exit node.
3. Sort the nodes in a list by nonincreasing order of $rank_u$ values.
4. **while** there are unscheduled nodes in the list **do**
5. **begin**
6. Select the first task n_i in the list and remove it.
7. Assign the task n_i to the processor p_j that minimizes the (EFT) value of n_i.
8. **end**

Figure 1. The HEFT Algorithm

1. Compute $rank_u$ for all nodes by traversing graph upward, starting from the exit node.
2. Compute $rank_d$ for all nodes by traversing graph downward, starting from the start node.
3. $|CP| = rank_u(n_s)$, where n_s is the *start* node.
4. **For** each node n_i do
5. **If** $(rank_d(n_i) + rank_u(n_i) = |CP|)$ then
6. n_i is a critical path node (CPN).
7. Select the critical-path-processor that minimizes $\sum_{n_i \in CPN} w_{i,j}$.
8. Initialize the priority-queue with the entry nodes.
9. **while** there is an unscheduled node in the priority-queue do
10. **begin**
11. Select the highest priority node from priority-queue,
12. which maximizes $rank_d(n_i) + rank_u(n_i)$.
13. **if** (n_i is a CPN) then
14. schedule n_i to critical-path-processor.
15. **else**
16. Assign the task n_i to the processor p_j which minimizes the (EFT) value of n_i.
17. Update the priority-queue with the successor(s) of n_i if they become ready-nodes.
18. **end**

Figure 2. The CPOP Algorithm

Mapping Heuristic (MH) Algorithm The MH Algorithm uses *static* upward ranks ($rank_u^s$) to assign priorities to the nodes. A ready node list is kept sorted according to the decreasing order of priorities. (For tie-breaking, the node with the largest number of immediate successors is selected.) With a noninsertion-based method, the processor that provides the minimum earliest finish time of a task is selected to run the task. After a task is scheduled, the immediate successors of the task are inserted into the list. These steps are repeated until all nodes are scheduled. The time complexity of this algorithm is $O(v^2 \times p)$ for v nodes and p processors.

Dynamic-Level Scheduling (DLS) Algorithm The DLS Algorithm assigns node priorities by using an attribute called *Dynamic Level* (DL) that is equal to $DL(n_i, p_j) = rank_u(n_i) - EST(n_i, p_j)$. (In contrast to the mean values, *median* values are used to compute the static upward ranks; and for the EST computation, the noninsertion method is used). At each scheduling step the algorithm selects the (ready node, available processor) pair that maximizes the dynamic level value. For heterogeneous environments the $\Delta(n_i, p_j)$ term is added to the dynamic level computation. The Δ value for a task on a processor is computed by the difference between the task's median execution time on all processors and its execution time on the current processor. The DLS Algorithm has an $O(v^3 p \times f(p))$ time complexity, where v is the number of tasks and p is the number of processors. The complexity of the function used for data routing to calculate the earliest start time is $O(f(p))$ [4].

Levelized-Min Time Algorithm. This is a two-phase algorithm. The first phase orders the tasks based

on their precedence constraints, i.e., level by level. This phase groups the tasks that can be executed in parallel. The second phase is a greedy method that assigns each task (level by level) to the "fastest" available processor as much as possible. A task in a lower level has higher priority for scheduling than a node in a higher level; within the same level, the task with the highest average computation cost has the highest priority. If the number of tasks in a level is greater than the number of available processors, the fine-grain tasks are merged into a coarse-grain task until the number of tasks is equal to the number of processors. Then the tasks are sorted in reverse order (largest task first) based on average computation time. Beginning from the largest task, each task will be assigned to the processor: a) that minimizes the sum of computation cost of the task and the communication costs with tasks in the previous layers; and b) that does not have any scheduled task at the same level. For a fully-connected graph, the time complexity is $O(p^2v^2)$ when there are v tasks and p processors.

5. Performance and Comparison

In this section we present the performance comparison of our algorithms with the related work, using the randomly generated task graphs and regular task graphs representing the applications. The following metrics were used to compare the performances of our proposed algorithms with the previous approaches.

- **Schedule Length Ratio (SLR)**. The SLR of an algorithm is defined as
  $SLR = \frac{makespan}{\sum_{n_i \in CP_{MIN}} \min_{j \in P}\{w_{i,j}\}}$
  where $makespan$ is the schedule length of the algorithm's output schedule. The denominator is the summation of computation costs of nodes on the CP_{MIN}. (For an unscheduled DAG, if the computation cost of each node n_i is set with $\{\min_{j \in P}\{w_{i,j}\}\}$, then the resulting critical path, CP_{MIN}, will be based on minimum computation values). The SLR value of any algorithm for a DAG cannot be less than one, since the denominator in the equation is the lower bound for completion time of the graph. The algorithm that gives the lowest SLR value of a DAG is the best algorithm with respect to performance (i.e., the one that gives the minimum overall execution time).

- **Speedup**. The speedup value of an algorithm is computed by dividing the overall computation time of the sequential execution to the makespan of the algorithm. (The sequential execution time is computed by assigning all tasks to a single processor that minimizes the total computation time of the DAG). Formally, it is defined as:
  $Speedup = \frac{min_{j \in P}(\sum_{n_i \in DAG} w_{i,j})}{makespan}$.

- **Running Time**. This is the average cost of each scheduling algorithm for obtaining the schedules of the given graphs on a Sun SPARC 10 workstation. The trade-offs between the performance (SLR value) and cost (running time) of scheduling algorithms were given in this section.

5.1. Generating Random Task Graphs

We developed an algorithm to generate random directed acyclic graphs that are used to evaluate the proposed scheduling algorithms. The input parameters required to generate a weighted random DAG are the following:

- Number of nodes (tasks) in the graph, v.

- Shape parameter of the graph, α. We assume that the height of a DAG is randomly generated from a uniform distribution with mean equal to $\alpha \times \sqrt{v}$. The width for each level in a DAG is randomly selected from a uniform distribution with mean equal to $\frac{\sqrt{v}}{\alpha}$. If $\alpha = 1.0$, then it will be a balanced DAG. A dense DAG (a shorter graph with high parallelism) can be generated by selecting $\alpha >> 1.0$. Similarly, if $\alpha << 1.0$, it will generate a longer DAG with a small parallelism degree.

- Out degree of a node, out_degree. One method is to use a fixed out_degree value for all nodes in a DAG as much as possible. Another alternative is to use a mean out_degree. Then, each node's out-degree will be randomly generated from a uniform distribution with mean equal to out_degree.

- Communication to computation ratio, CCR. CCR is the ratio of the average communication cost to the average computation cost. If a DAG's CCR value is low, it can be considered as a computation-intensive application; if it is high, it can be considered as a communication-intensive application.

- Average computation cost in the graph, avg_comp. Computation costs are generated randomly from a uniform distribution with mean equal to avg_comp. Similarly, the average communication cost is derived as $avg\_comm = CCR \times avg\_comp$.

- Range percentage of computation costs on processors, β. A high β value causes significant difference of a node's computation cost among the processors; a very low β value means that the expected execution times of a node on any given processors are almost equal. If the average computation cost of a node n_i is $\overline{w_i}$, then the computation cost of n_i on any processor p_j will be randomly selected from the range, $\overline{w_i} \times (1 - \frac{\beta}{2}) \leq w_{i,j} \leq \overline{w_i} \times (1 + \frac{\beta}{2})$, where β is the range percentage.
- Number of available processors in the system, num_pe.

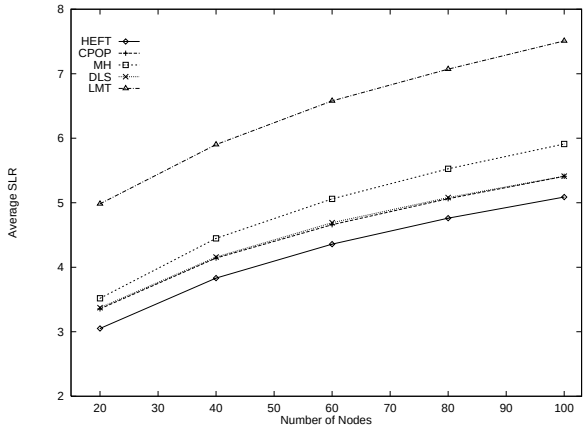
To compare the scheduling algorithms presented, we used random graphs as a test suite. The input parameters of the directed acyclic graph generation algorithm were set with the following values:

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

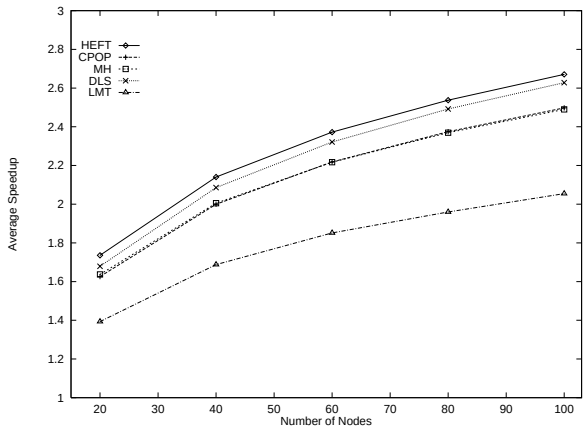
These combinations give 2250 different DAG types. Since we generated 25 random DAGs for each DAG type, the total number of DAGs used in our comparison study is around 56250. The following part gives the algorithm rankings for several graph parameters. Each ranking starts with the best algorithm and ends with the worst one, with respect to a given parameter.

Performance Study with Respect to Graph Size

The performances (SLR values) of algorithms were compared with respect to different graph sizes in Figure 3(a). The SLR value for each graph size is the average SLR value of 11250 different graphs that were generated randomly with different CCR, α, β, and out_degree values when the number of available processors was equal to 4. The performance ranking of the algorithms will be {HEFT, CPOP, DLS, MH, LMT }. The average SLR value of HEFT on all generated graphs will be better than CPOP by 7%, DLS Algorithm by 8%, MH Algorithm by 16% and LMT Algorithm by 52%. The average speedup curve with respect to the number of nodes is given in Figure 3(b). The average speedup ranking of the algorithms is {HEFT, DLS, (CPOP=MH), LMT}. We repeated these tests for the case when the number of processors was equal to 10. In this case, although the average SLR values were lower than in the previous case, they gave the same performance ranking of the algorithms.



(a)



(b)

Figure 3. (a) Average SLR (b) Average Speedup with Respect to Graph Size

Figure 4 shows the average running time of each algorithm. From this figure it can be concluded that the HEFT algorithm is the fastest and the DLS algorithm is slowest among the given algorithms. When the average running time of the algorithms on all graphs is computed by combining the results of different graph sizes, the HEFT Algorithm will be faster than the CPOP Algorithm by 10%, the MH Algorithm by 32%, the DLS Algorithm by 84%, and the LMT Algorithm by 48%.

Performance Study with Respect to Graph Structure

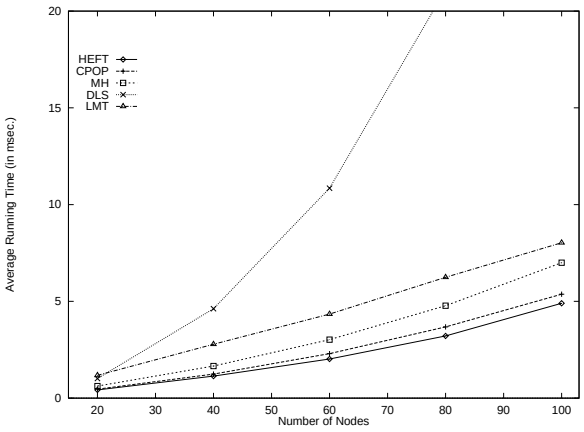
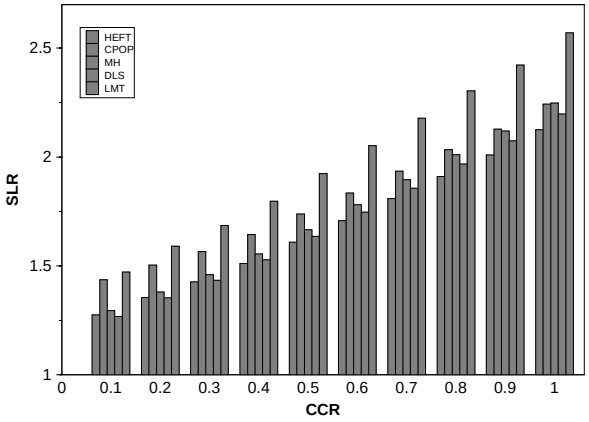


Figure 4. Average Running Time of the Algorithms with Respect to Graph Size

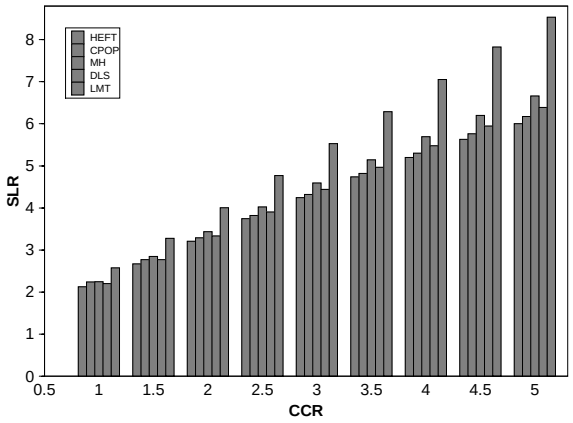
When α (the shape parameter of a graph) is equal to 0.5, the generated graphs have longer depths with a low parallelism degree. If the average SLR values of the algorithms are compared when $\alpha = 0.5$, the performance of the HEFT Algorithm is better than the CPOP Algorithm by 8%, the MH Algorithm by 12%, the DLS Algorithm by 6%, and the LMT Algorithm by 40%. When α is equal 1.0, the average depth of the graph and average width of each layer in the graph will be approximately equal. For this case, the average SLR value of the HEFT Algorithm will be better than the CPOP Algorithm by 7%, the MH Algorithm by 14%, the DLS Algorithm by 7%, and the LMT Algorithm by 34%. Similarly, if $\alpha = 2.0$, the average width will be approximately equal to four times the average depth, which will come up with short and dense graphs. For this case, the HEFT Algorithm will be better than the CPOP Algorithm by 6%, the MH Algorithm by 15%, the DLS Algorithm by 8%, and the LMT Algorithm by 31%. For all three α values, the HEFT Algorithm gives the best performance among the five algorithms.

Performance Study with Respect to CCR

Figure 5 gives the average SLR of the algorithms when the CCR value in $[0.1, 1.0]$ and $[1.0, 5.0]$ with number of processors is equal to 10. When $CCR \geq 0.25$, the HEFT Algorithm outperforms the other algorithms. If $CCR \leq 0.2$, the DLS algorithm gives a better performance than the HEFT Algorithm. 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}.



(a)



(b)

Figure 5. Average SLR of the Algorithms with Respect to CCR: (a) $0.1 \leq CCR \leq 1.0$ **(b)** $1.0 \leq CCR \leq 10.0$

5.2. Applications

In this section we present a performance comparison of the scheduling algorithms based on Gaussian elimination and Fast Fourier Transformation (FFT). For the Gaussian elimination algorithm, we did the analytical work to set approximately the computation costs of the nodes and communication costs of the edges. The number of nodes in the Gaussian elimination algorithm can be characterized by the input matrix size. For FFT, the computation cost of each node is randomly set from a normal distribution with mean equal to an assigned average computation cost. The communication costs are set randomly from a normal distribution with mean equal to the multiplication of CCR and average communication cost. The number of nodes in an FFT task graph can be computed in terms of number of input points (i.e., order of the FFT).

In order to provide different processor speeds, the *computing power weights* [18] of the processors were randomly set from a given range. The *computing power weight* of each processor P_j, (φ^j), is a number that shows the ratio of CPU speed of P_j to the CPU speed of the fastest processor in the system. In order to set the computation cost of each task on processors, the computation cost of the node on a base processor (P_{base}) is set (either randomly as in FFT or analytically as in the Gaussian elimination). Then the computation cost for each other processor P_i is computed by $W(T_j, P_k) = \frac{\varphi^{base}}{\varphi^k} \times W(T_j, P_{base})$. For the fastest processor, P_F, $\varphi^F = 1$.

In the experiments the range of the computing power weights can be $[0.8-1.0]$, $[0.6-1.0]$, $[0,4-1.0]$ or $[0.2-1.0]$. The first range is for a domain in which the computational powers of processors are almost equal; however, at the last set ($[0.2-1.0]$) the fastest processor can be up to five times faster than the slowest one. We also varied the task graph granularities by varying the CCR values in the range $\{0.1, 0.5, 1.0, 2.0, 5.0, 10.0\}$.

Gaussian elimination. Figure 6(a) gives the sequential program for the Gaussian elimination algorithm [1, 17]. The data-flow graph of the algorithm for the special case of $n = 5$, where n is the dimension of the matrix, is given in Figure 6(b). The number of nodes in the task graph of this algorithm is roughly $\frac{n^2+n-2}{2}$. Each $T_{k,k}$ represents a pivot column operation, and each $T_{k,j}$ represents an update operation. The node computation cost function for any task $T_{k,j}$ is equal to $W_{k,j} = 2 \times (n-k) \times w$, where $1 \leq k \leq j \leq n$ (w is the average execution time of either an addition and multiplication, or of a division).

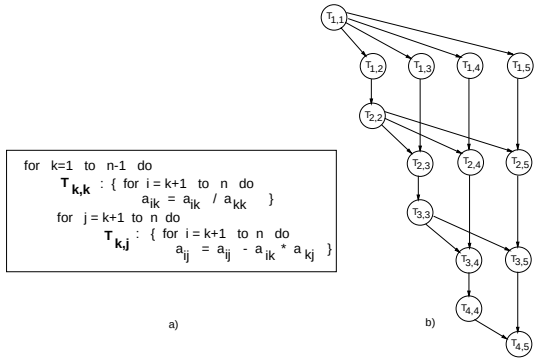
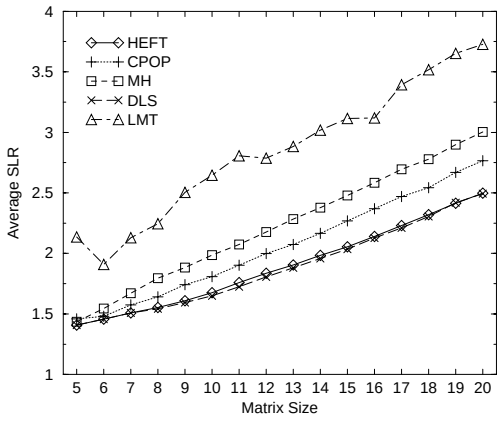


Figure 6. (a) The Gaussian elimination Algorithm (kji version) (b) The Task graph for a Matrix of Size 5

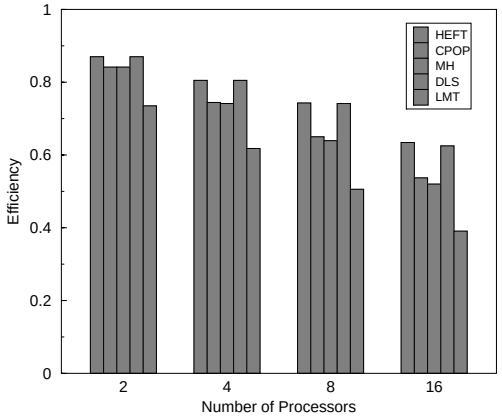
The edge communication cost function between any edge is $C_{k,j} = (n-k) \times b$, where b is the transmission rate. Each Gaussian elimination task graph has one critical path (CP), which has the maximum number of nodes covered as compared to the other paths. In Figure 6(b), the critical path is $T_{1,1}T_{1,2}T_{2,2}T_{2,3}T_{3,3}T_{3,4}T_{4,4}T_{4,5}$.

Figure 7(a) gives the average SLR values of the algorithms at various matrix sizes. The performances of the HEFT and DLS algorithms were the best of all. The performance ranking of the algorithms for Gaussian elimination graphs is {(HEFT = DLS), CPOP, MH, LMT}. As in the results of other applications, the SLR values of all scheduling algorithms slightly increase if the matrix size is increased. Increasing the matrix size causes more nodes not to be on the critical path, which causes increases in the makespan.

Figure 7(b) gives the efficiency comparison of the algorithms for the Gaussian elimination graphs of about 1250 nodes (i.e., the matrix size is 50) at various numbers of processors. Efficiency is the ratio of the speedup value to the number of processors used. The HEFT and DLS algorithms have greater 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. For all scheduling algorithms, increasing the number of processors reduces efficiency because the matrix size (thus the number of nodes) is fixed. Additionally, an increase in the number of processors causes fewer nodes than the number of processors at some levels of the graph, which will decrease the utilization of the processors.



(a)



(b)

Figure 7. (a) Average SLRs of Scheduling Algorithms at Various Graph Sizes for Gaussian Elimination Graph (b) The Efficiency Comparison of the Algorithms

Figure 8 gives the average running time of each algorithm for the Gaussian elimination graph for a various number of processors. Although the DLS algorithm gives as good performance as the HEFT algorithm for Gaussian elimination graphs, it is the slowest algorithm. (For Gaussian elimination task graphs when matrix size is 50 with 16 processors, the DLS algorithm takes 16.2 times longer time than the HEFT algorithm to schedule tasks). The running time of the HEFT algorithm is comparable to the MH algorithm, but it is greater than the LMT algorithm. The cost ranking of the algorithms (starting from the lowest) for the Gaussian elimination graphs is {LMT,(HEFT, MH),CPOP,DLS}. Although the LMT Algorithm is a higher-complexity algorithm than the others except for the DLS algorithm (see Figure 4), it gives the best running time for Gaussian elimination graphs due to the fact that half of the levels of a Gaussian elimination graph has a single node, which decreases the cost of the LMT algorithm.

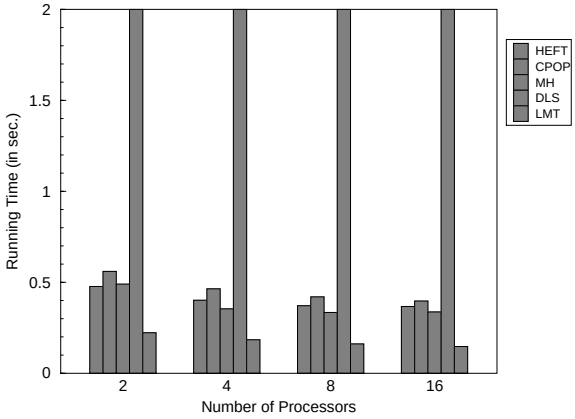


Figure 8. Average Running Time for Each Algorithm to Schedule Gaussian Elimination Graphs

Fast Fourier Transformation The recursive, one-dimensional FFT Algorithm [15, 16] and its task graph is given in Figure 9. A is an array of size n, which holds the coefficients of the polynomial; 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 Figure 9(b) can be divided into two parts: the nodes above the dashed line are the recursive call nodes (RCNs), and the ones below the line are butterfly operation nodes (BONs). For an input vector of size n, there are $2 \times n - 1$ RCNs and

$n \times \log_2 n$ BONs. (We assume that $n = 2^m$ for some integer m). Each path from the start node to any of the exit nodes in an FFT task graph is a critical path because the computation cost of nodes in any level are equal, and the communication costs of edges between two consecutive levels are equal.

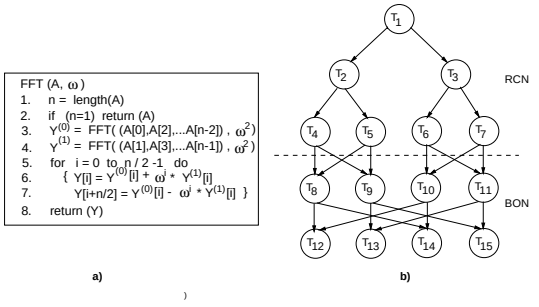
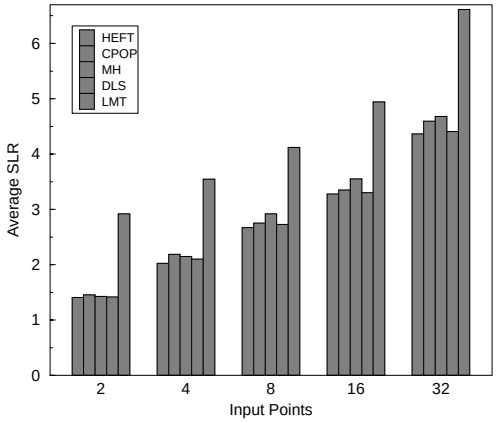
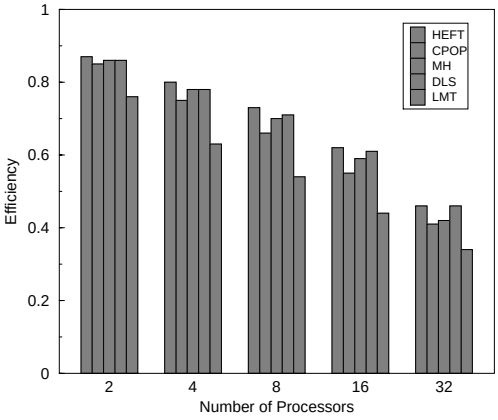


Figure 9. (a) FFT Algorithm (b) The Generated DAG of FFT with Four Points.

Figure 10(a) shows the average SLR values of scheduling algorithms for FFT graphs for various numbers of input points when the number of processors is equal to six. The HEFT algorithm outperforms the other algorithms. The performance ranking of the algorithms is {HEFT, DLS, CPOP, MH, LMT}. Figure 10(b) shows the efficiency values obtained for each of the algorithms when there are 64 data points. The HEFT and DLS algorithms result in good schedules for all cases. The running time comparisons of the algorithms, with respect to number of input points and with respect to number of processors, were shown in Figure 11. For FFT graphs, the DLS and LMT Algorithms are high-cost scheduling algorithms, whereas the HEFT, CPOP, and MH Algorithms are low-complexity scheduling algorithms. Based on these results it can be concluded that the HEFT algorithm is the best algorithm in terms of performance and cost.



(a)

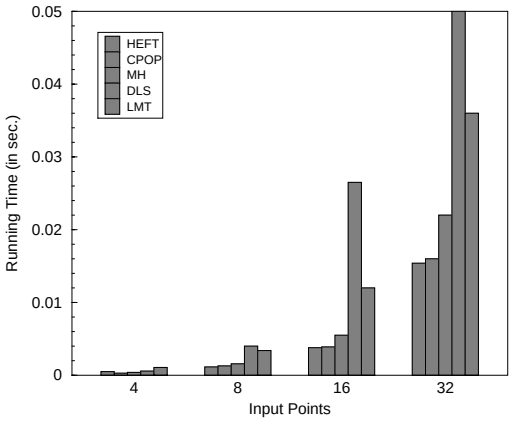


(b)

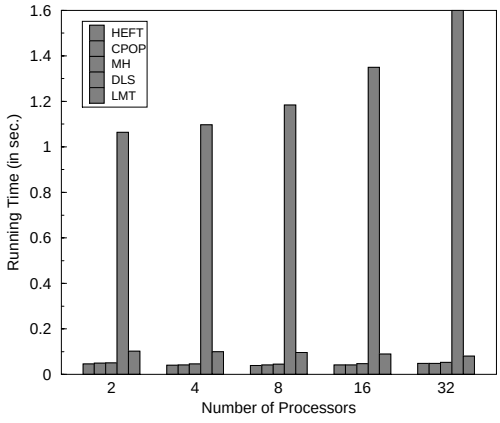
Figure 10. (a) Average SLRs of Scheduling Algorithms at Various Graph Sizes for FFT Graph (b) Efficiency Comparison of the Algorithms

6. Conclusion and Future Work

In this paper we have proposed two task scheduling algorithms (the HEFT Algorithm and the CPOP Algorithm) for heterogeneous processors. For the generated random task graphs and the task graphs of for selected applications, the HEFT algorithm outperforms the other algorithms in all performance metrics, i.e., average schedule length ratio (SLR), speedup, and time-complexity. Similarly, the CPOP Algorithm is better than, or at least comparable to, the existing algorithms. Both algorithms perform more stably than



(a)



(b)

Figure 11. Running Times of the Scheduling Algorithms for FFT Graphs

the others in terms of scheduling quality and running time.

We are extending our algorithms to improve their performance at specific CCR ranges and computing power weight ranges. Additionally, we plan to add low-cost, local-search techniques to improve the scheduling quality of our algorithms. Future work will include different techniques for ordering the ready tasks and extensions for the processor selection phase.

References

[1] M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message Passing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 330–343, July 1990.

[2] Y. Kwok and I. Ahmad, "Dynamic Critical-Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, May 1996.

[3] E. S. H. Hou, N. Ansari, and H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 113–120, Feb. 1994.

[4] G. C. Sih and E. A. Lee, A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, pp. 175–186, Feb. 1993.

[5] H. El-Rewini and T. G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 138–153, 1990.

[6] H. Singh, A. Youssef, "Mapping and Scheduling Heterogeneous Task Graphs Using Genetic Algorithms," *Heterogeneous Computing Workshop*, 1996.

[7] I. Ahmad and Y. Kwok, "A New Approach to Scheduling Parallel Programs Using Task Duplication," *Proc. of Int'l Conference on Parallel Processing*, vol. II, pp. 47–51, 1994.

[8] M. A. Palis, J. Liou, and D. S. L. Wei, "Task Clustering and Scheduling for Distributed Memory Parallel Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, 46–55, Jan. 1996.

[9] M. Iverson, F. Ozguner, G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environments," *Proc. of Heterogeneous Computing Workshop*, 93–100, 1995.

[10] P. Shroff, D. W. Watson, N. S. Flann, and R. Freund, "Genetic Simulated Annealing for Scheduling Data-Dependent Tasks in Heterogeneous Environments," *Proc. of Heterogeneous Computing Workshop*, 1996.

[11] T. Yang and A. Gerasoulis, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, 276–291, 1992.

[12] T. Yang and A. Gerasoulis. "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, Sept. 1994.

[13] Y. Kwok and I. Ahmad, "A Static Scheduling Algorithm Using Dynamic Critical Path For Assigning Parallel Algorithms onto Multiprocessors," *Proc. of Int'l Conference on Parallel Processing*, vol. II, pp. 155–159, 1994.

[14] L. Wang, H. J. Siegel, and V. P. Roychowdhury, "A Genetic-Algorithm-Based Approach for Task Matching and Scheduling in Heterogeneous Computing Environments," *Proc. of Heterogeneous Computing Workshop*, 1996.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

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

[17] M. Cosnard, M. Marrakchi, Y. Robert, D. Trystram, "Parallel Gaussian Elimination on an MIMD Computer," *Parallel Computing*, vol. 6, pp. 275-295, 1988.

[18] Y. Yan, X. Zhang, Y. Song, "An Effective Performance Prediction Model for Parallel Computing on Non-dedicated Heterogeneous NOW," *Journal of Parallel and Distributed Computing*, vol. 38, pp. 63-80, 1996.

Biographies

Haluk Topcuoglu is a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at Syracuse University, New York, USA. His research interests include task scheduling techniques in heterogeneous environments, metacomputing and cluster computing issues, parallel and distributed programming, and web technologies. He received his B.S. and M.S. degrees in 1991 and 1993, respectively, in computer engineering from Bogazici University, Istanbul, Turkey. Mr. Topcuoglu is a member of ACM, IEEE, IEEE Computer Society, and the Phi Beta Delta Honor Society.

Salim Hariri is currently a Professor in the Department of Electrical and Computer Engineering at The University of Arizona. Dr. Hariri received his Ph.D in computer engineering from University of Southern California in 1986, an M.S. degree from The Ohio State University, in 1982, and B.S. in Electrical Engineering from Damascus University, in 1977. His current research focuses on high performance distributed computing, design and analysis of high speed networks, benchmarking and evaluating parallel and distributed systems, and developing software design tools for high performance computing and communication systems and applications. Dr. Hariri is the Editor-In-Chief for the Cluster Computing Journal. He served as the Program Chair and the General Chair of the IEEE International Symposium on High Performance Distributed Computing (HPDC).

Min-You Wu received the M.S. degree from the Graduate School of Academia Sinica, Beijing, China, and the Ph.D. degree from Santa Clara University, California. Before he joined the Department of Electrical and Computer Engineering, University of Central Florida, where he is currently an Associate Professor, he has held various positions at University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, and State University of New York at Buffalo. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 70 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a member of ACM and a senior member of IEEE. He is listed in International Who's Who of Information Technology.