Improving Static Task Scheduling in Heterogeneous and Homogeneous Computing Systems*

Chih-Hsueh Yang†, PeiZong Lee†, and Yeh-Ching Chung‡
†Institute of Information Science, Academia Sinica
‡Department of Computer Science, National Tsing Hua University

Abstract

In this paper, we present a heuristic algorithm that improves the performance of static task scheduling. Our algorithm is based on the list-scheduling mechanism. For the listing phase, we use existing techniques to generate partial-order task sequences based on critical-path-first ordering, critical-task-first ordering, and their hybrids. For the scheduling phase, we propose a task-duplication algorithm with a look-ahead technique, so that the complexity of the new algorithm does not increase. The experiment results show that our algorithm outperforms other algorithms for any feasible task sequences with respect to the average execution times and the average scheduling length ratios.

1 Introduction

A system is heterogeneous if its connected computers are different; otherwise, the system is homogeneous. In general, an application can be represented by a weighted directed-acyclic task graph. The algorithm for finding optimal solutions for the multiple-processor scheduling problem has been shown to be NP-complete [3, 4, 13]. Various heuristic algorithms that have been proposed are capable of finding sub-optimal solutions [6, 7, 8, 10, 17, 18, 19, 20]. These heuristics are categorized into several classes, such as list-based algorithms [5, 6, 12, 17, 20], clustering algorithms [4, 9, 11, 19], and duplication-based algorithms [1, 18, 16, 18]. Among these algorithms, list-based scheduling algorithms are generally regarded as having a good cost-performance trade-off because of their low cost and acceptable results [2, 4, 13, 14].

In [17], a fast load balancing algorithm assigns all ready tasks (whose parent tasks have been scheduled) in the same step. In [20], two list-based algorithms are presented, one of which favors tasks with the earliest finishing time, and the other favors tasks on the critical paths. In [6], a list-based algorithm (HCNFD) based on finding critical nodes for heterogeneous systems is proposed. The algorithm generates a partial-order task sequence (called HCN ordering in this paper) based on critical nodes on the critical paths and allows tasks to be duplicated. The experimental studies in [6] show that the HCNFD algorithm outperforms the above three algorithms presented in [17, 20].

In this paper, we adopt a list-scheduling mechanism whose performance depends on the task sequence and the scheduling technique. We studied three popular sequences, the first favors the critical path, the second favors the critical task, and the third is a hybrid of the first two. In the scheduling phase, we use a task-duplication algorithm as a basis, and also implement the HCNFD algorithm proposed in [6] for comparison.

Based on the above-mentioned scheduling algorithm, we propose a look-ahead method to improve the performance of list scheduling. This method takes the next task in the sequence into consideration when assigning an immediate task. Through experimental studies, we found that the look-ahead mechanism improves the performance of various scheduling methods in average cases.

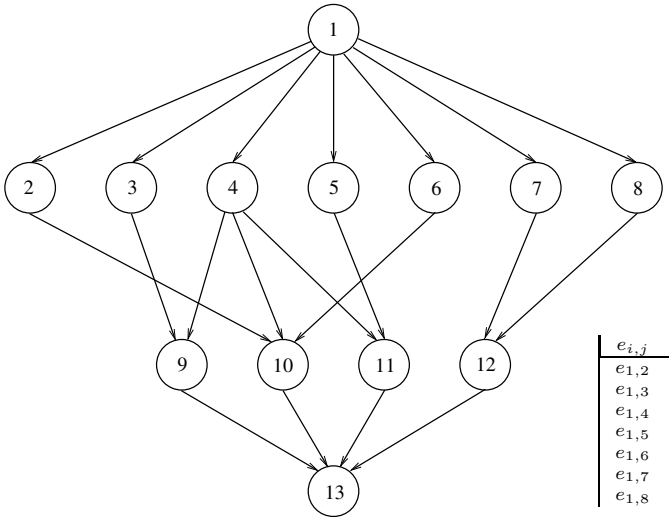
The remainder of this paper is organized as follows. Section 2 defines the problem, including the task graphs and the computation/communication costs. Section 3 describes our proposed task scheduling algorithms. Section 4 presents the experiment results. Then in Section 5, we present our conclusion.

2 Problem Definition

In this section, we follow the notations presented in [6] to define the models of heterogeneous and homogeneous computing systems, the model of the application used for static scheduling, and the scheduling objective. We use the example shown in Figure 1 to illustrate our method.

A **heterogeneous computing system** is a set P of p heterogeneous processing elements (PEs) connected in a fully

*This work was partially supported by the NSC under Grants NSC 94-2213-E-001-013 and NSC 95-2221-E-001-003. Corresponding author: PeiZong Lee, leepe@iis.sinica.edu.tw.



$$S = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

$$R = \begin{pmatrix} 0 & 2 & 4 \\ 4 & 0 & 6 \\ 5 & 3 & 0 \end{pmatrix}$$

$$W = \begin{pmatrix} 22 & 15 & 17 \\ 15 & 25 & 5 \\ 22 & 5 & 9 \\ 19 & 19 & 16 \\ 16 & 20 & 24 \\ 16 & 14 & 9 \\ 22 & 5 & 9 \\ 19 & 19 & 16 \\ 15 & 12 & 21 \\ 9 & 21 & 3 \\ 5 & 6 & 25 \\ 15 & 12 & 21 \\ 10 & 17 & 24 \end{pmatrix}$$

$e_{i,j}$	$\mu_{i,j}$	$\overline{c_{i,j}}$	$e_{i,j}$	$\mu_{i,j}$	$\overline{c_{i,j}}$	$e_{i,j}$	$\mu_{i,j}$	$\overline{c_{i,j}}$
$e_{1,2}$	4	18	$e_{2,10}$	4	18	$e_{7,12}$	5	22
$e_{1,3}$	2	10	$e_{3,9}$	5	22	$e_{8,12}$	3	14
$e_{1,4}$	1	6	$e_{4,9}$	3	14	$e_{9,13}$	4	18
$e_{1,5}$	2	10	$e_{4,10}$	6	26	$e_{10,13}$	2	10
$e_{1,6}$	3	14	$e_{4,11}$	5	22	$e_{11,13}$	3	14
$e_{1,7}$	2	10	$e_{5,11}$	2	10	$e_{12,13}$	4	18
$e_{1,8}$	1	6	$e_{6,10}$	3	14			

Figure 1. Description of a sample application in a heterogeneous computing system with three PEs.

connected topology. It is assumed that:

- Any PE can execute a task and communicate with other PEs at the same time because of overlapping computation and communication time.
- Once a PE has started a task, it continues without interruption. Then, after completing the execution, it immediately sends the corresponding output data to all of its child tasks in parallel.

The communication costs per transferred unit (of several bytes) between any two PEs are stored in a matrix R of size $p \times p$, where $R_{i,j}$ represents the communication cost of sending a unit of data from $\texttt{PE}_i$ to $\texttt{PE}_j$. The communication startup costs of PEs are given in a p-dimensional vector S, where S_i represents the startup cost of $\texttt{PE}_i$.

An **application** is represented by a weighted *directed-acyclic graph* (DAG) $G(V, E, W)$, where:

V is the set of v nodes; and each node $v_i \in V$ represents an application task, which is an indivisible code segment that must be executed sequentially on the same PE.

W is a $v \times p$ computation cost matrix in which each $w_{i,j}$ gives the estimated time to execute task v_i on $\texttt{PE}_j$.

E is the set of communication edges. The directed edge $e_{i,j}$ connects nodes v_i and v_j, where node v_i is called the parent node and node v_j is called the child node. This implies that v_j cannot start until v_i finishes and sends its data to v_j. Each edge is associated with a value $\mu_{i,j}$, which represents the amount of output data transmitted from task v_i to task v_j (in a unit of several bytes).

A task node without any parent is called an *entry task*, and a task node without any child is called an *exit task*. If there are two or more entry (exit) tasks, they may be connected to a zero-cost pseudo entry (exit) task with zero-cost edges, which does not affect the schedule.

The communication cost of the edge $e_{i,j}$, which represents the cost of transferring $\mu_{i,j}$ units (of several bytes) of data from task v_i (scheduled on p_m) to task v_j (scheduled on p_n), is defined as:

$$c_{i,j} = S_m + R_{m,n} \cdot \mu_{i,j},$$

where S_m is the communication startup time of p_m, $\mu_{i,j}$ is the amount of data transferred from task v_i to task v_j, and $R_{m,n}$ is the communication time per transferred unit (of several bytes) from p_m to p_n (in seconds/unit).

The *average earliest start time* $\texttt{AEST}(v_i)$ of node v_i can be computed recursively by traversing the DAG downward starting from the entry node $v_{\texttt{entry}}$

$$\texttt{AEST}(v_i) = \texttt{max}_{v_m \in \texttt{pred}(v_i)}\{\texttt{AEST}(v_m) + \overline{w_m} + \overline{c_{m,i}}\},$$

where $\texttt{pred}(v_i)$ is the set of immediate predecessors of v_i and $\texttt{AEST}(v_{\texttt{entry}}) = 0$.

The *average latest start time* $\texttt{ALST}(v_i)$ of node v_i can be computed recursively by traversing the DAG upward starting from the exit node $v_{\texttt{exit}}$

$$\texttt{ALST}(v_i) = \texttt{min}_{v_m \in \texttt{succ}(v_i)}\{\texttt{ALST}(v_m) - \overline{c_{i,m}}\} - \overline{w_i},$$

where $\texttt{succ}(v_i)$ is the set of immediate successors of v_i and $\texttt{ALST}(v_{\texttt{exit}}) = \texttt{AEST}(v_{\texttt{exit}})$.

The average execution cost of task v_i is computed as follows:

$$\overline{w_i} = (1/p) \sum_{j=1}^{p} w_{i,j},$$

v_i	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
$\overline{w_i}$	18	15	12	18	20	13	12	18	16	11	12	16	17
$\texttt{AEST}(v_i)$	0	36	28	24	28	32	28	24	62	69	64	62	96
$\texttt{ALST}(v_i)$	0	42	28	30	40	48	28	30	62	75	70	62	96

Table 1. $\overline{w_i}$, $\texttt{AEST}(v_i)$, and $\texttt{ALST}(v_i)$ for the example in Figure 1.

and the average communication cost of sending data from task v_i to task v_j is computed as follows:

$$\overline{c_{i,j}} = \overline{S} + \overline{R} \cdot \mu_{i,j},$$

where $\overline{S} = (1/p) \sum_{i=1}^{p} S_i$ is the average communication startup cost over all PEs, and $\overline{R} = (1/p^2) \sum_{i=1}^{p} \sum_{j=1}^{p} R_{i,j}$ is the average communication cost per transferred unit over all PEs. Table 1 shows $\overline{w_i}$, $\texttt{AEST}(v_i)$, and $\texttt{ALST}(v_i)$ for the example in Figure 1; $\overline{c_{i,j}}$ can be seen in Figure 1.

A **homogeneous computing system** is a special case of a heterogeneous computing system, in which all PEs have the same execution speed, and the communication overhead between any two PEs is the same. For instance, all entries $w_{i,j}$ in matrix W for $1 \leq j \leq p$ are the same; all entries S_j in vector S for $1 \leq j \leq p$ are the same; and all entries $R_{i,j}$ in matrix R for $1 \leq i \neq j \leq p$ are the same.

The main **objective** of task scheduling is to minimize the scheduling length ($makespan$), while satisfying the tasks' dependencies.

3 Proposed Algorithms

A list-scheduling algorithm is comprised of two phases: the first is the listing phase, which generates a partial-order task sequence, where tasks conform to dependency relations; the second is the scheduling phase, which assigns tasks in a generated partial-order sequence to PEs. In Section 3.1 we review some existing methods for generating various partial-order sequences; in Section 3.2 we present an existing algorithm for scheduling tasks on PEs; and in Section 3.3 we present our new scheduling algorithm.

3.1 Listing Phase

The goal of the listing phase is to find a partial-order sequence that conforms with the dependency relations between tasks in order to schedule each task. We use $\texttt{ALST}(v_i)$ to represent the priority of task v_i; the smaller value of $\texttt{ALST}(v_i)$ has a higher priority. During the generation of a task sequence, we say that a task is *ready* if it has not been generated, but all its parent tasks have been generated. We now review three methods.

The **Critical-Path-First** method (CPF) tries to select tasks on the critical path prior to other tasks. The flavor of this method also can be seen in [5, 6, 17, 20]. A task v_i is on the critical path if $\texttt{ALST}(v_i) = \texttt{AEST}(v_i)$. The main concept of this method is twofold. First, tasks on a critical path should be executed as early as possible; second, consecutive tasks on a critical path should be assigned to the same PE to avoid communication overhead. The ordering created by CPF looks just like performing a **depth-first search** on the (priority-based) task graph from the entry task. For the application shown in Figure 1, the CPF ordering is: 1, 3, 4, 9, 7, 8, 12, 5, 11, 2, 6, 10, 13.

The **Critical-Task-First** method (CTF) attempts to select the most critical task when it is ready. The flavor of this method also can be seen in [17, 20]. A task v_i is the most critical if $\texttt{ALST}(v_i)$ is minimal among all ready tasks. The main concept of this method is also twofold. First, as usual, the most critical task should be executed as early as possible; second, it may result in more ready tasks, so that when assigning these ready tasks to PEs, their execution times and communication times may overlap. The ordering created by CTF is just like performing a **breadth-first search** on the (priority-based) task graph from the entry task. For the application shown in Figure 1, the CTF ordering is: 1, 3, 7, 4, 8, 5, 2, 6, 9, 12, 11, 10, 13.

A **Hybrid** method takes advantage of both CPF and CTF. The CPF ordering seems to result in better performance for a task graph with a long and thin shape; while CTF ordering seems to result in better performance for a task graph with a wide body. Various characteristics of the task graph may influence the generation of an ordering, such as the number of ready tasks, the height of the remaining task graph, and the number of PEs. In this paper we only present one heuristic. If the number of ready tasks is greater than or equal to 1.5 multiplied by the number of PEs, then we adopt the CTF method; otherwise we adopt the CPF method. For the application shown in Figure 1, the Hybrid ordering is: 1, 3, 7, 4, 8, 5, 2, 6, 10, 9, 12, 11, 13.

There are other methods of generating task sequences; however, in this paper it is not our purpose to generate new sequences. Instead, we present a new algorithm in Section 3.3, which can improve task scheduling for sequences generated by any method.

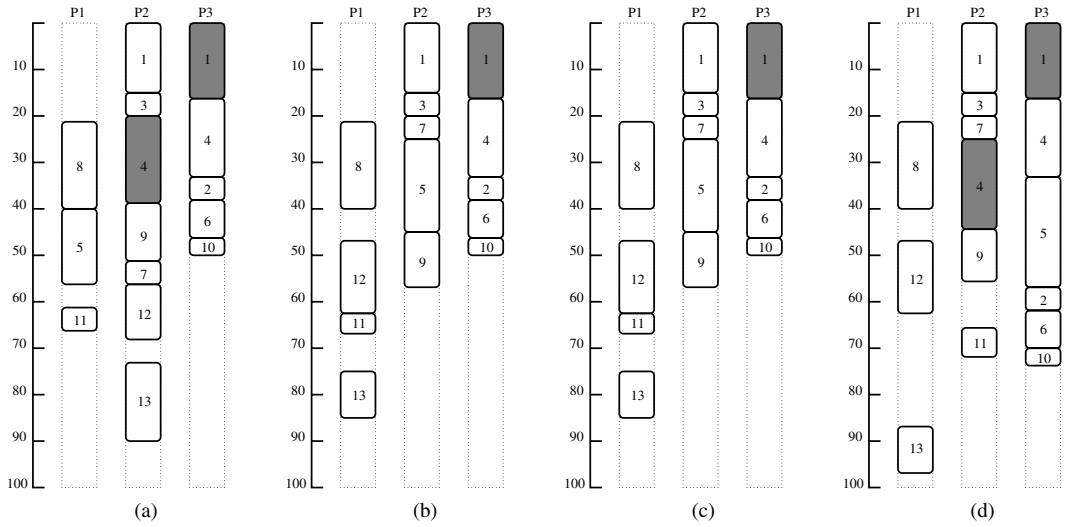


Figure 2. Scheduling of the task graph in Figure 1 with: (a) DUP using CPF ordering (makespan = 90), (b) DUP using CTF ordering (makespan = 85), (c) DUP using Hybrid ordering (makespan = 85), and (d) HCNFD using HCN ordering (makespan = 97).

3.2 Scheduling Phase

In the scheduling phase, an algorithm is used to assign tasks in the sequence (generated in the listing phase) to appropriate PEs. For every assignment, a task v_j can start to run in $\mathtt{PE}_m$ if all its parent tasks v_i have finished in some PEs and have transferred the necessary data (say, $\mu_{i,j}$ units of data) to $\mathtt{PE}_m$. We present a *duplication* algorithm, which will serve as the basis for our new look-ahead algorithm in Section 3.3. The flavor of this duplication algorithm also can be seen in [1, 15, 16, 18]. In effect, any scheduling algorithm can be used as the basis of our look-ahead algorithm.

A duplication algorithm

- While picking each task v_i in turn from the task sequence generated in the listing phase, we assign the task to be executed to the PE with the earliest finishing time provided that, if necessary, we can replicate at most one parent task with respect to task v_i in the same PE.
- If both PEs report the same finishing time for an assignment, we choose the PE that did not duplicate v_i's parent task, or required the shortest execution time for task v_i.
- If task v_i is assigned to be executed in $\mathtt{PE}_m$, where one of v_i's parent tasks has just been replicated and this parent task has only one child task v_i, then we can remove this parent task from its originally assigned PE (to save execution time).

Figure 2 shows the scheduling results of the duplication algorithm (DUP) for CPF ordering, CTF ordering, and the Hybrid ordering of the application in Figure 1; the gray tasks are the duplicated tasks. We also list the scheduling result using the HCNFD algorithm [6] for comparison, where the HCNFD algorithm adopts the generated HCN ordering: 1, 3, 7, 4, 9, 8, 12, 5, 11, 2, 6, 10, 13.

3.3 A Look-ahead Algorithm

In heterogeneous computing systems, different PEs may have different capabilities to run different tasks. For instance, suppose that task $v_i(v_j)$ is more suitable to be executed in $\mathtt{PE}_m(\mathtt{PE}_n)$ because $w_{i,m}(w_{j,n})$ is minimum for $1 \leq m(n) \leq p$, respectively, and that task v_i is ahead of task v_j in the task sequence. In addition task v_i can start at an earlier time t_1 and finish at an earlier time t_2 in $\mathtt{PE}_n$ in comparison that task v_i only can start at a later time t_3 and finish at a later time t_4 in $\mathtt{PE}_m$, where $t_1 < t_3$, $t_2 < t_4$, and $(t_2 - t_1)$ is much larger than $(t_4 - t_3)$. However, using the duplication algorithm in Section 3.2, it is possible that task $v_i(v_j)$ is assigned to $\mathtt{PE}_n(\mathtt{PE}_m)$ which is not the most effective PE for executing task $v_i(v_j)$. Therefore, in this section we consider the assignment of two consecutive tasks (in the task sequence) simultaneously, but we only fix one assignment and leave the other task to the next assignment.

A look-ahead algorithm

The basic assignment algorithm is the same as the duplication algorithm in Section 3.2. However, in the αth step, we consider the task left from the $(\alpha - 1)$th assignment, say v_i, and the $(\alpha + 1)$th task in the task sequence, say v_j, where

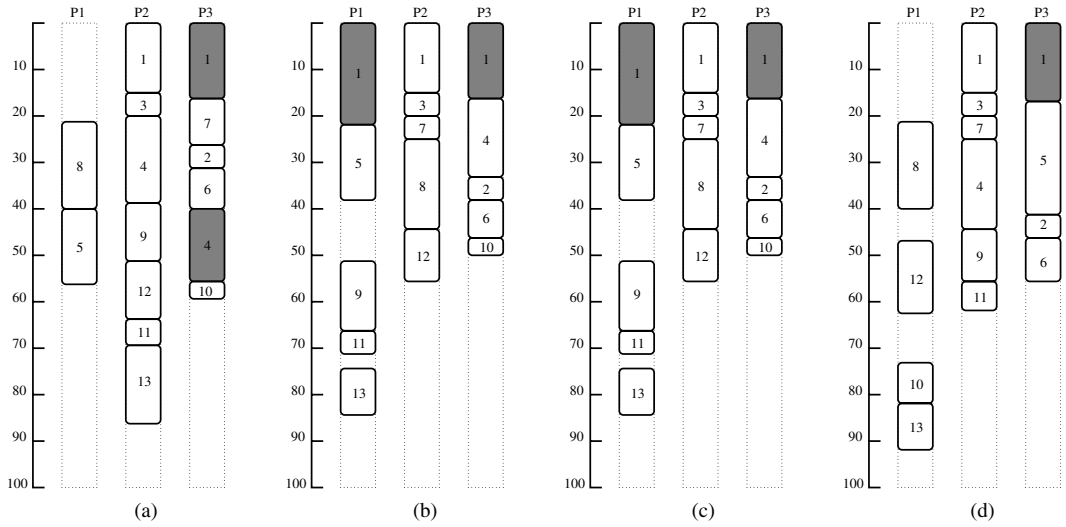


Figure 3. Scheduling of the task graph in Figure 1 with: (a) DUP/LA using CPF ordering (makespan = 86), (b) DUP/LA using CTF ordering (makespan = 84), (c) DUP/LA using Hybrid ordering (makespan = 84), and (d) HCNFD/LA using HCN ordering (makespan = 92).

$1 \leq \alpha \leq v$, the task left from the 0th assignment is the first task $v_{\texttt{entry}}$ in the sequence, and the $(v+1)$th task in the sequence is a null task. We consider two cases depending on whether v_i is a parent of v_j.

Case 1: v_i and v_j are independent. We want to compare four finishing times. First, we assign v_i, where the finishing time is A, and then we assign v_j whose finishing time is B. Second, we assign v_j first and the finishing time is C, after which we assign v_i whose finishing time is D.

If $\texttt{max}\{A, B\} \leq \texttt{max}\{C, D\}$, we fix the assignment of v_i and leave v_j to the next assignment.

If $\texttt{max}\{A, B\} > \texttt{max}\{C, D\}$, we fix the assignment of v_j and leave v_i to the next assignment.

Case 2: v_i is a parent of v_j. We want to compare $(p+1)$ finishing times of v_j, where the first finishing time is derived by assigning v_i first and then assigning v_j. The remaining p finishing times are derived by assigning both v_i and v_j in $\texttt{PE}_m$, for $1 \leq m \leq p$, respectively.

We choose the case in which the finishing time of v_j is the earliest, and fix the assignment of v_i, but leave v_j to the next assignment.

Figure 3 shows the scheduling results of the look-ahead algorithm based on the duplication algorithm (DUP/LA) according to the CPF ordering, CTF ordering, and the Hybrid ordering of the application in Figure 1. We also list the scheduling results using the look-ahead algorithm based on the HCNFD algorithm (HCNFD/LA) for comparison, where a HCN ordering 1, 3, 7, 4, 9, 8, 12, 5, 11, 2, 6, 10, 13 generated by the HCNFD algorithm is used as the task sequence. We observe that for all these four cases, the look-ahead algorithm performs better than those not using the look-ahead technique as shown in Figure 2.

3.4 Algorithm Complexity Analysis

In the task graph, let the number of task nodes be v and the number of edges be e. Let the number of heterogeneous PEs be p, and let $p \leq v$.

- The complexity of computing `AEST` and `ALST` of all nodes is $O(e+v)$.
- For each assignment step of v_i in the duplication algorithm in Section 3.2, it is necessary to compute the start time for executing v_i in each PE, and search for the earliest available time slot in that PE. As v_i only needs to know the information about its parents and there are at most $v+p$ available time slots among all PEs, where $p \leq v$, the time complexity of each assignment step of v_i is $O(p \cdot \texttt{degree}(v_i) + v)$, where $\texttt{degree}(v_i)$ is the number of edges incident to v_i. Thus, the complexity of assigning all tasks in the sequence is

$$O(p \cdot \sum_{i=1}^{v} \texttt{degree}(v_i) + v^2) = O(p \cdot e + v^2).$$

- For each αth assignment step in the look-ahead algorithm in Section 3.3, the αth and $(\alpha+1)$th tasks (v_i and v_j) are considered, where $1 \leq \alpha \leq v$ and the

$(v+1)$th task is a null task. If v_i and v_j are independent, the complexity of this step is proportional to $(2p(\texttt{degree}(v_i) + \texttt{degree}(v_j)) + 4v)$.

If v_i is a parent of v_j, there are two conditions: 1) if v_i and v_j are assigned to different PEs, then the complexity is proportional to $(p(\texttt{degree}(v_i) + \texttt{degree}(v_j)) + 2v)$; and 2) if v_i and v_j are assigned to the same PE, then the complexity is also proportional to $(p(\texttt{degree}(v_i) + \texttt{degree}(v_j)) + 2v)$. Therefore, the complexity of assigning all tasks in the sequence is proportional to

$$(2p \cdot (\sum_{i=1}^{v} \texttt{degree}(v_i) + \sum_{j=1}^{v} \texttt{degree}(v_j)) + 4v^2),$$

say, $O(p \cdot e + v^2)$.

Clearly the complexities of the duplication algorithm and the look-ahead algorithm are the same. Note that adding the look-ahead mechanism to the duplication algorithm does not increase the complexity penalty.

4 Experimental Studies

This section presents a performance comparison of the duplication algorithm (DUP) in Section 3.2, the HCNFD algorithm [6], and the look-ahead algorithm in Section 3.3 based on either the duplication algorithm (DUP/LA) or the HCNFD algorithm (HCNFD/LA). We use a set of randomly generated application graphs as the test suite, and adopt the metrics suggested in [6] for the performance evaluation.

4.1 Comparison Metrics

The comparison of the different algorithms is based on the following three metrics.

Makespan: The scheduling length is called makespan; the shorter the makespan the better.

Scheduling length ratio (SLR): The time taken to execute tasks on a critical path is the lower bound of the schedule length. To normalize the schedule length to the lower bound, the SLR is defined as

$$\texttt{SLR} = \texttt{makespan}/(\sum_{v_i \in \texttt{CP}_{\texttt{min}}} \texttt{min}_{1 \le j \le p}\{w_{i,j}\}).$$

The denominator is the sum of the minimum computation costs of the tasks on a critical path $\texttt{CP}_{\texttt{min}}$ (without including any communication costs); therefore, the smaller the SLR the better.

Quality results of schedules: The number of times that an algorithm produced a better or worse result, or scheduling equal quality compared to every other algorithm, is counted in the experiments.

In [6], *speedup* is also used as a metric. Suppose that the execution time required to execute all tasks in the most efficient PE is called $\texttt{SINGLE}_{\texttt{min}}$; then, $\texttt{speedup} = \texttt{makespan}/\texttt{SINGLE}_{\texttt{min}}$. Speedup has the same merits as makespan or SLR; therefore, we do not consider speedup in this presentation.

4.2 Random Graph Generator

A random graph generator is implemented to generate application graphs with various characteristics. The generator requires the following input parameters.

- The number of tasks v.
- The number of edges e.
- Heterogeneity factor β, where a temporary w_i is generated randomly for each task and a randomly selected β from a β set is used to compute $w_{i,j} = \beta \cdot w_i$ for each v_i on $\texttt{PE}_j$.
- Communication to computation ratio (CCR), which is defined as the ratio of the average communication cost $((1/e)(\sum_{e_{i,j} \in E} \overline{c_{i,j}}))$ to the average computation cost $((1/(vp))(\sum_{i=1}^{v} \sum_{j=1}^{p} w_{i,j}))$.

In this presentation, the input parameters were selected from the following values for four heterogeneous PEs: $v \in \{12, 13, 14, 15, 16\}$, $0.9v \le e \le 1.5v$, $\beta \in \{1.0, 2.0, 3.0\}$, and $\texttt{CCR} \in \{0.5, 1.0, 2.0\}$. We generated sets of 100 application graphs for each v, β, and CCR from the above list, a total of 4,500 application graphs.

4.3 Performance Results

Figure 4 shows the performance results for four heterogeneous PEs. We apply the HCNFD algorithm [6], the duplication algorithm (DUP), and the look-ahead algorithm (LA) using lists of tasks generated by HCN ordering (generated by the HCNFD algorithm), CPF ordering, CTF ordering, and Hybrid ordering.

Figures 4(a) and (b) show that, by using the look-ahead technique, the number of times that the HCNFD algorithm and the DUP algorithm perform better is more than not using it for every task ordering sequence. Figures 4(c) and (d) show that the average execution times (makespans) and the average SLRs of using the look-ahead technique are better

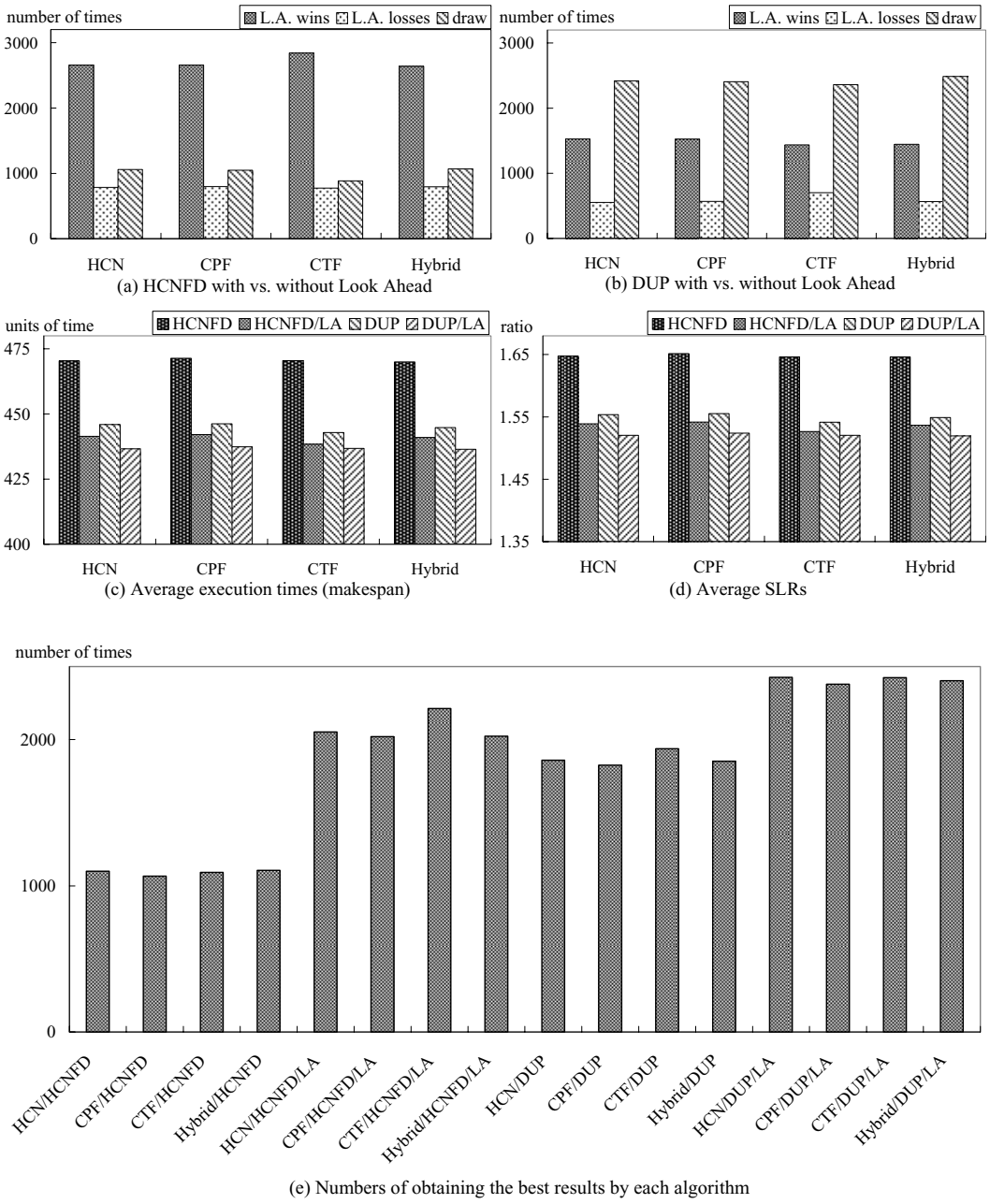


Figure 4. For four heterogeneous PEs, number of wins, losses, or draws by applying or not applying the look-ahead technique for the (a) HCNFD and (b) DUP algorithms, and (c) the average execution times (makespans), (d) the average SLRs, and (e) the number of best results by each algorithm.

(shorter or smaller) than not using it for every task ordering sequence. Figure 4(e) shows that the look-ahead technique has the best chance of achieving the best execution time among the tested algorithms.

We also found similar trends for homogeneous systems such that, on average, using the look-ahead technique performs better than not using it.

5 Conclusion

In this paper, we have studied how to schedule task graphs in heterogeneous and homogeneous systems. We presented list-based scheduling algorithms, and generated partial-order task sequences based on CPF ordering, CTF ordering, and a hybrid of both types. We found that none of the existing task-scheduling algorithms can outperform other algorithms based on any task sequences generated by fixed ordering. We applied a look-ahead mechanism to existing task duplication algorithms, and, on average, achieved a better performance than algorithms that do not use a look-ahead mechanism for any partial-order task sequences. The proposed look-ahead algorithm has the same complexity as existing task duplication algorithms; thus, it can be used at compiling time to schedule static task graphs in both heterogeneous and homogeneous systems.

References

[1] I. Ahmad and Y.-K. Kwok. A new approach to scheduling parallel programs using task duplication. In *Proc. International Conf. on Parallel Processing*, pages 47–51, 1994.

[2] Y.-C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputing '92: Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 512–521, 1992.

[3] B. Demiroz and H. R. Topcuoglu. Static task scheduling with a unified objective on time and resource domains. *The Computer Journal*, 49(6):731–743, June 2006.

[4] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling dags on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[5] T. Hagras and J. Janecek. A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments. In *Proc. International Conf. on Parallel Processing Workshops*, pages 149–155, Oct. 2003.

[6] T. Hagras and J. Janecek. An approach to compile-time task scheduling in heterogeneous computing systems. In *Proc. International Conf. on Parallel Processing Workshops*, pages 182–189, Aug. 2004.

[7] T. Hagras and J. Janecek. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. In *Proc. International Parallel and Distributed Processing Symposium*, 2004.

[8] T. Hagras and J. Janecek. A static task scheduling heuristic for homogeneous computing environment. In *Proc. Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2004.

[9] M. Hakem and F. Butelle. Dynamic critical path scheduling parallel programs onto multiprocessors. In *Proc. International Parallel and Distributed Processing Symposium*, 2005.

[10] J. Han and Q. Li. A novel static task scheduling algorithm in distributed computing environment. In *Proc. International Parallel and Distributed Processing Symposium*, 2004.

[11] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Trans. Parallel Distributed Syst.*, 7(4):506–521, May 1996.

[12] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[13] J.-C. Liou and M. A. Palis. A comparison of general approaches to multiprocessor scheduling. In *Proc. International Parallel Processing Symposium*, pages 152–156, 1997.

[14] C. L. McCreary, A. A. Khan, J. J Thompson, and M. E. McArdle. A comparison of heuristics for scheduling DAGs on multiprocessors. In *Proc. International Parallel Processing Symposium*, pages 446–451, April 1994.

[15] K. Oh-Han and A. Dharma. S3MP: A task duplication based scalable scheduling algorithm for symmetric multiprocessor. In *Proc. International Parallel and Distributed Processing Symposium*, pages 451–456, 2000.

[16] G. Park, B. Shirazi, and J. Marquis. DFRN: A new approach for duplication based scheduling for distributed memory multiprocessor system. In *Proc. International Conf. on Parallel Processing*, pages 157–166, 1997.

[17] A. Radulescu and A. J. C. van Gemund. Fast and effective task scheduling in heterogeneous systems. In *Heterogeneous Computing Workshop*, pages 229–238, 2000.

[18] S. Ranaweera and D. P. Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Proc. International Parallel and Distributed Processing Symposium*, pages 445–450, May 2000.

[19] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans. Parallel Distributed Syst.*, 4(2):175–187, 1993.

[20] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Syst.*, 13(3):260–274, March 2002.