Critical-Task Anticipation Scheduling Algorithm for Heterogeneous and Grid Computing

Ching-Hsien Hsu[1], Ming-Yuan Own[1], and Kuan-Ching Li[2]

[1] Dept. of Computer Science and Information Engr. Chung Hua University, Taiwan
chh@chu.edu.tw
[2] Dept. of Computer Science and Information Engr. Providence University, Taiwan
kuancli@pu.edu.tw

Abstract. The problem of scheduling a weighted directed acyclic graph (DAG) to a set of heterogeneous processors to minimize the completion time has been recently studied. The NP-completeness of the problem has instigated researchers to propose different heuristic algorithms. In this paper, we present an efficient Critical-task Anticipation (*CA*) scheduling algorithm for heterogeneous computing systems. The *CA* scheduling algorithm introduces a new task prioritizing scheme that based on urgency and importance of tasks to obtain better schedule length compared to the Heterogeneous Earliest Finish Time algorithm. To evaluate the performance of the proposed algorithm, we have developed a simulator that contains a parametric graph generator for generating weighted directed acyclic graphs with various characteristics. We have implemented the *CA* algorithm along with the *HEFT* scheduling algorithm on the simulator. The *CA* algorithm is shown to be effective in terms of speedup and easy to implement.

1 Introduction

The demand for powerful computing to solve a large application has emerged in recent years. Some parallel architecture, such as multiple computers, or multiple processor system, that employ numerous processors interconnected by high-speed network to achieve superior performance than use a single computer. Because the diverse quality among that processors (computers) or some special requirement, like exclusive function, memory access speed, or the customize I/O devices, etc.; the tasks have distinct execution time on different processors (computers) and it named heterogeneous computing system.

The purpose of such system is to drive processors cooperate to get the application (an application consists of tasks) done quickly. Therefore, one of the key factors is how to schedule individual task among processors to minimize execution time or maximize processor utilization and so on. The primary scheduling methods can be classified into two categories: dynamic scheduling and static scheduling. In dynamic algorithm, it executes redistribution of tasks between processors during run-time, expect to balance computational load, and reduce processor's idle time. On the contrary, in static algorithm, it assigns tasks to processors at the compile time, attempt to minimize the entire completion time, and satisfy the precedence of tasks [6, 14]. When the

C. Jesshope and C. Egan (Eds.): ACSAC 2006, LNCS 4186, pp. 95 – 108, 2006.
© Springer-Verlag Berlin Heidelberg 2006

information of an application which predict tasks execution time, the message size of communication among tasks, and tasks dependences are known a priori at the compile-time, it called static model [6, 14], thus, schedule analysis must be done before run time.

A Direct Acyclic Graph (DAG) [2] is used for modeling parallel applications which consists of several independent tasks. The nodes of DAG correspond to tasks and the edges of which indicate the precedence constraints between tasks. In addition, the weight of an edge represents communication cost among tasks. Each node is given a computation cost and it is represented by a computation costs matrix. Figure 1 depicts an example of DAG and the computation cost matrix. Moreover, we consider that each task can be executed on a single processor only and tasks are non-preemptable. A task n_j is a successor (predecessor) of task n_i if there exists an edge from n_i to n_j (from j to i) in the graph. The task has precedence constraint, that is, only if the predecessor n_i completes its execution and then its successor n_j receives the *messages* from n_i, the successor n_j can start its execution.

The scheduling problem has been widely researched in heterogeneous system where the computational ability of processors is different and the processors communicate over an underlying network. Many researchers had proposed articles on the subject. The scheduling problem in general is proved to be NP-complete, so the desire of optimal scheduling can lead to higher scheduling overhead. The negative result motivates the requirement for heuristic approaches to solve the scheduling problem. A comprehensive survey about static scheduling algorithms is given in [14]. The authors of [14] have shown that the heuristic-based algorithms can be classified into a variety of categories, such as clustering algorithms, duplication-based algorithms, and list-scheduling algorithms.

The keynote of clustering algorithms is a mapping of the tasks onto n clusters. Each task in a cluster must execute in the same processor. A nonlinear clustering is that at least one cluster contains two independent tasks, otherwise it called linear clustering. It iterates clustering steps while no improvements in the scheduling length can be obtained. The requirement of unbound processors was a disadvantage and it causes the algorithm to work badly in practice [3, 16]. With the auxiliary of some cluster merge steps, the problem was solved [7, 8], although the approach is expensive.

The duplication-based algorithms [1, 9, 11] are another different skills. Those algorithms utilize the duplicate technique which duplicates some critical tasks (i.e. the parent tasks) on the same or another processor so that reduce the communication cost. When duplication of the execution of tasks occurs in processors, it will result in an increase in the space complexity since data must be duplicated too.

The list-scheduling algorithms [10, 13, 14, 15] divided the approach into two independent parts, list phase and processor-selection phase. In the first part, list phase, they used heuristic method to give the task a priority and then according as the priority, make an arrangement for the task set. In the second part, processor-selection phase, they used the result of the first part to select the most suitable processor for the task assignment. Our Critical-task Anticipation (*CA*) algorithm belongs to this classification. This typical method is superior to the others because it is easier to practice, lower complexity, and good performance.

In this paper, our proposed algorithm uses the following scheduling system model. There are P fully connected heterogeneous processors in the system. The processors

communicate over an underlying communication network which is contention-free. The main intent of this problem is to minimize the schedule length (schedule length also called *makespan*). Our proposed algorithm takes advantage of some graph attributes used by heterogeneous earliest-finish-time (*HEFT*) algorithm [14], and furthermore we came up with a novel idea to improve the performance. In the *HEFT* algorithm, it detects the critical path length of a given node. To do so, it uses *critical score*, i.e., as the name implies, an accumulative value that are computed recursively travels along the graph upward, starting from the exit node. In the literature, the authors exhibited the brilliant performance as compared with the Dynamic Level Scheduling Algorithm [13], the Levelized-Min Time Algorithm [5], and the Mapping Heuristic Algorithm [12]. Our algorithm is similar to the *HEFT* algorithm, except that we use a critical-task anticipation skill. We add a simple modification to make significant improvements in schedule length as well as speedup of the application.

The rest of this paper is organized as follows: Section 2 introduces the scheduling system and problem formulation. Section 3 presents the definitions used in our proposed algorithm. In section 4, we discuss details of the *CA* scheduling algorithm and give a simple comparison to the *HEFT* algorithm. Section 5 shows the simulation results. Finally, in Section 6, some concluding remarks are made.

2 Preliminaries

2.1 Heterogeneous Scheduling System

As mentioned in section 1, the heterogeneous computing architecture is a set of heterogeneous processors $P = \{p_k\!: k = 1\!: p\}$ connected in a fully connected topology, where $p = |P|$. We also assume that:

1) There is no network contention between any arbitrary processors.
2) Computation and communication can be worked simultaneously because of the separated I/O.
3) Tasks are non-preemptable. In other words, once a task is assigned to a processor, it starts execution and finishes to its completion.
4) After accomplish the task's execution, the task have to send operational result to all immediate successor of it instantly.

W is an $n \times p$ matrix in which $w_{i,j}$ indicates estimated computation time to execute task n_i on processor p_j. The mean value of task n_i is calculated as follow:

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

The communication cost depends on the size of message and communication latency of processors. A $p \times p$ matrix T is structured to represent data transfer rate among processors. Latency of processors is given in a p-dimensional vector V. The communication cost of transferring data from task n_i (execute on processor p_m) to task n_j (execute on processor p_n) is denoted by $c_{i,j}$ and can be calculated by the following equation

$$c_{i,j} = V_m + \frac{Message_{i,j}}{T_{m,n}}, \tag{2}$$

Where:

V_m is the latency of processor p_m,

$Message_{i,j}$ is the size of message from task n_i to task n_j,

$T_{m,n}$ is data transfer rate from processor p_m to processor p_n.

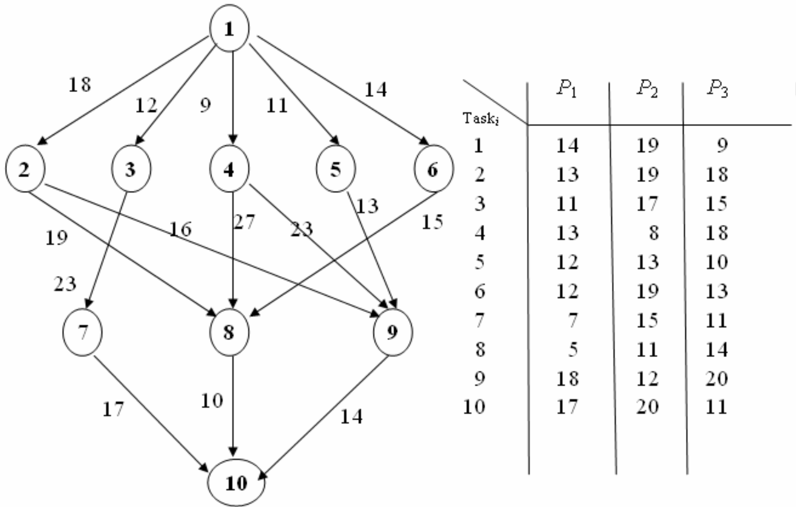
In static scheduling model, it is usually to use the mean value of communication cost to simplify the presentation in a given DAG (as shown in Fig. 1). The mean value of communication cost between tasks n_i and n_j can be formulated by the following equation,

$$\overline{c_{i,j}} = \overline{V} + \frac{Message_{i,j}}{\overline{T}} . \tag{3}$$

Where:

$\overline{V}$ is the average latency of processors.

$\overline{T}$ is the average transfer rate.



$Task_i$	P_1	P_2	P_3
1	14	19	9
2	13	19	18
3	11	17	15
4	13	8	18
5	12	13	10
6	12	19	13
7	7	15	11
8	5	11	14
9	18	12	20
10	17	20	11

Fig. 1. An example of Direct Acyclic Graph (DAG) and the computation cost matrix

2.2 Problem Formulation

The application can be represented by a Directed Acyclic Graph (DAG), G = (V, E, C), where V = $\{n_j: j = 1: v\}$ is the set of nodes and $v = |V|$, E = $\{e_{i,j} = \langle n_i, n_j \rangle\}$ is the set of communication edges and $e = |E|$; C is the set of edge communication costs. In the DAG model, each node indicates least indivisible unit, in other words, each node must be executed on a processor from beginning to end. Each edge $\langle n_i, n_j \rangle$ is a direct arc on which n_i is the immediate predecessor and n_j is the immediate successor. There is precedence relationship between tasks, namely task n_j takes it's turn to prepare for starting execution after the n_i has finished it's execution and n_j receive the essential

message from n_i. The weight of edge <n_i, n_j > indicates the average communication cost between n_i and n_j.

The node without any inward edge is called *entry node* n_{entry}, and a node without any outward edge is called *exit node* n_{exit}. In general, it is supposed that the application has only one *entry node* and one *exit node*. If the actual application claims more than one *entry* (*exit*) *node*, we can accede a zero-cost fake *entry* (*exit*) *node* with zero-cost edge.

The goal of scheduling problem is minimizing the total execution time of the application. If there are more than one *exit tasks*, we consider that the latest completion task is the ending of the application. In other word, we want to shorten the schedule length as far as possible.

3 Definitions

The list scheduling algorithm is broadly distinguished into list phase and processor-selection phase. In this section, we give some definitions that will be used in both *CA* and *HEFT* algorithms.

3.1 Parameters for List Phase

Definition 1: In the list phase, the *Critical Score* of the task n_{exit} denoted by $CS(n_{exit})$ is defined as $CS(n_{exit}) = \overline{w_{exit}}$, where $\overline{w_{exit}}$ is the average computation cost of task n_{exit}.

Definition 2: $CS(n_i) = \overline{w_i} + \max_{n_j \in suc(n_i)} (\overline{c_{i,j}} + CS(n_j))$, where $\overline{w_i}$ is the average computation cost of task n_i, $\overline{c_{i,j}}$ is the average communication cost of edge <n_i, n_j>, and $suc(n_i)$ is the set of immediate successors of task n_i.

3.2 Parameters for Processor-Selection Phase

In the processor-selection phase, the algorithm exploits a partial schedule to meet the minimum schedule length. There is an intuitional idea to calculate the *finish time* (*FT*) of task n_j that will be executed on processor p_k, then we can select the minimum finish time from the calculated results and determine which processor is chosen to execute the task n_j. In such approach, each processor p_k will maintain a list of tasks, *task-list*(p_k), keeps the latest status of tasks correspond to the $EFT(n_i, p_k)$, the earliest finish time of task n_i that is assigned on processor p_k.

Recall having been mentioned above that the application represented by DAG must satisfy the precedence relationship. Taking into account the sequence of tasks which are assigned on the processors, a task n_j can intend to execute on a processor p_k only if its all immediate predecessors send the essential messages to n_j and n_j successful receives all these messages. Thus, the latest message arrive time of node n_i on processor p_k, denoted by $LMAT(n_j, p_k)$, is evaluated by the following equation,

$$LMAT(n_j, p_k) = \max_{n_i \in pred(n_j)} \left(EFT(n_i) + \overline{c_{i,j}}\right) \tag{4}$$

where $pred(n_j)$ is the set of immediate predecessors of task n_j. Note that if tasks n_i and n_j are assigned to the same processor, $\overline{c_{i,j}}$ is assumed to be zero because it is negligible.

Definition 3: The n_{entry} has no inward arc, therefore for the task n_{entry}, $LMAT(n_{entry}, p_k) = 0$, for all $k = 1$ to p.

Definition 4: The *start time* of task n_j executed on processor p_k is denoted as $ST(n_j, p_k)$. The determination of start time aims to search available time slot on processor p_k that is large enough to execute task n_j (i.e., length of time slot > $w_{j,k}$). Note that the search of available time slot is started from $LMAT(n_j, p_k)$.

Definition 5: The *finish time* of task n_j completes its execution on processor p_k is denoted as $FT(n_j, p_k)$ and calculated by the following equation,

$$FT(n_j, p_k) = ST(n_j, p_k) + w_{j,k} \tag{5}$$

Definition 6: The *earliest finish time* of task n_j completes its execution is denoted as $EFT(n_j)$ and determined by the following equation,

$$EFT(n_j) = \underset{p_k \in P}{Min}\{FT(n_j, p_k)\} \tag{6}$$

Definition 7: According to definition 6, if the *EFT* of task n_j is determined upon task n_j executed on processor p_t, then the target processor of task n_j is denoted by $TP(n_j)$, and $TP(n_j) = p_t$.

4 The Proposed Scheduling Algorithm

In this section, we first present a new scheduling algorithm, the critical-task anticipation algorithm (outlined in Figure 2) which will be operated in the heterogeneous scheduling system. The proposed scheduling algorithm will be verified beneficial for the readers while we delineate a sequence of the algorithm and show some example scenarios. In the rest of this section, we will review the *HEFT* algorithm which is the best known list-scheduling algorithm and provide some different viewpoint between both algorithms.

4.1 The Critical-Task Anticipation Scheduling Algorithm

The $CS(n_i)$ is known as the maximal summation of scores including the average computation cost and communication cost from task n_i to the exit task, that is, $CS(n_i)$ is the longest length of critical path. Therefore, the magnitude of the task's critical score is regarded as the decisive factor when we arrange the priority. In the *HEFT* algorithm, it sorts the tasks in L by non-increasing order of critical scores. This method seems good intuitively that it provides suitable priorities for the tasks. In this study, we propose an improving scheduling heuristic, the critical-task anticipation scheduling algorithm (*CA*). The origin of the *CA* algorithm is owing to the following three observations.

Observation 1: The processors are heterogeneous, namely, there are variations in execution cost from processor to processor for each task. Different processor assignments for tasks result in a different computational cost. In that event, we always

wish to give the task n_i which has large average computation cost higher priority. This can aid the task n_i to get chance to reduce the finish time.

<u>Observation 2</u>: Except for the *entry task*, each task has to receive the essential messages from its immediate predecessors. In other words, a task will be in waiting state when it does not collect complete message yet. For this reason, we emphasize the importance of the last arrival message such that the succeeding task (node) can start its execution earlier. Therefore, it is imperative to give the predecessor who sends the last arrival message higher priority. This can aid the task to get chance to advance the start time.

<u>Observation 3</u>: If a task n_i is inserted into the front of the scheduling-list, it occupies advantage position. Namely, n_i has higher probability to reduce its finish time. Consequently, the start time of $suc(n_i)$ can be advanced with higher probability.

According to the above observations, we have a different viewpoint on the importance of a key task, the *critical-task* is defined as following.

<u>Definition 8</u>: A task n_i is a critical-task of task n_j, denoted as $CT(n_j)$, iff n_i is not inserted into scheduling list L yet and $CS(n_i) = \underset{n_k \in pred(n_j)}{Max}(CS(n_k))$.

Our viewpoint differs from the majority of literatures in terms of task prioritizing. In most algorithms, their thought is to schedule high critical score task first (even the estimation of critical scores in these algorithms are different). In our approach, the *CA* algorithm prioritizes the task n_i according to the influence of task n_i, which effects the successors of n_i (Observation 2) and devotes to lead to an accelerated chain (Observation 1). In short, our scheme is not only prioritizing tasks by the importance (i.e., critical score) but also prioritizing tasks by the urgent among tasks.

```
Begin:
1. Input the information of DAG and matrix.
/* List Phase */
2. Construct an empty scheduling-list L which is FIFO.
3. Calculate CS(n_i) for task n_i, ∀ n_i ∈ V.
4. Prioritize the tasks into L by CA procedure.
   //CA procedure is shown in figure 3.
   /* In the HEFT algorithm, tasks in L are sorted by
     non-increasing order according to critical
     scores */
/* Processor Selection Phase*/
5. While L is not empty do
6.       Remove task n_i from L.
7.       Compute LMAT(n_i, p_k), ST(n_i, p_k), FT(n_i, p_k) for all k = 1 to p.
8.       Determine EFT(n_i), EST(n_i).
9.       Assign task n_i to processor TP(n_i)
10.      Modify the task-list (TP(n_i)).
11.Endwhile
End
```

Fig. 2. The Proposed Critical-Task Anticipation Algorithm

4.2 Details and Example

The procedure of Critical-task Anticipation is outlined in Fig. 3. It maintains the following data structures: a scheduling list L which is first-in first-out, an auxiliary stack S, a temporary container C and an array of Boolean called queue vector (QV). $QV[n_i]$ = true indicates that task n_i has queued into L. $QV[n_i]$ = false indicates that task n_i has not yet queued into L.

We now perform a running trace of the *CA* algorithm. Let's consider again the example shown in figure 1, which has ten tasks. These tasks will be executed on three fully-connected heterogeneous processors. According to this DAG, the *critical scores* of tasks can be evaluated by definitions 1 and 2. We proceed to the computation of critical scores from the n_{exit} by bottom-up fashion. For example, for the exit node n_{10}, the $CS(n_{10})$=16, and for node n_8, $CS(n_8)=10 + max(10 + CS(n_{10})) = 10 + max(26) = 36$. We start to examine the procedure of critical-task anticipation algorithm which is illustrated in Fig. 3. The step by step execution sequence is given below.

Initially, QV = [F, F, F, F, F, F, F, F, F, F], S is empty, L is empty, where F is *false* and T is *true*. The index is the serial number of the task, from 1 to 10.

1) Push n_{10} on stack S. S = [n_{10}].
2) S is not empty, begin the while loop (Fig. 3, Line 5).
3) Pop n_{10}, predecessors of task n_{10} are n_7, n_8, n_9. Since the condition of QV at line 7 isn't satisfied, it then goes to the next stage, searching $CT(n_{10})$ at line 11 and resulting C = {n_7, n_8, n_9}. Finally, S = [n_9, n_7, n_8, n_{10}]. Note that $CT(n_{10}) = n_9$, which is pushed on the top of S, and n_9 has the highest priority than the other tasks in stack S.
4) Peek at n_9 (top of stack), then S = [n_4, n_2, n_5, n_9, n_7, n_8, n_{10}] (after lines 11-20 in CA procedure are processed).
5) Peek at n_4, then S = [n_1, n_4, n_2, n_5, n_9, n_7, n_8, n_{10}].
6) Peek at n_1, note that n_1 is entry node, so it follows lines 7~10, S = [n_4, n_2, n_5, n_9, n_7, n_8, n_{10}], L = [n_1] and set QV[n_1] = T.
7) Peek at n_4, because $pred(n_4) = \{n_1\}$, we then check QV[n_1] and have QV[n_1] = T. This implies that $pred(n_4)$ are inserted into L. Therefore, S = [n_2, n_5, n_9, n_7, n_8, n_{10}], L= [n_1, n_4] and set QV[n_4] = T (Lines 7-10).
8) Peek at n_2, S = [n_5, n_9, n_7, n_8, n_{10}], L = [n_1, n_4, n_2] and set QV[n_2] = T.
9) Peek at n_5, S = [n_9, n_7, n_8, n_{10}], L = [n_1, n_4, n_2, n_5] and set QV[n_5] = T.
10) Peek at n_9, $pred(n_9) = \{n_2, n_4, n_5\}$. Since QV = [T, T, F, T, T, F, F, F, F, F], the condition "all QV[n_i] are true, $n_i \in pred(n_j)$" at line 7 is satisfied. We then have S = [n_7, n_8, n_{10}], L = [n_1, n_4, n_2, n_5, n_9] and set QV[n_9] = T.
11) Peek at n_7, then S = [n_3, n_7, n_8, n_{10}].
12) Peek at n_3, then S = [n_7, n_8, n_{10}], L = [n_1, n_4, n_2, n_5, n_9, n_3] and set QV[n_3] = T.
13) We omit the rest process until only task n_{10} remains in stack S. When task n_{10} is popped, S becomes empty and L = [n_1, n_4, n_2, n_5, n_9, n_3, n_7, n_6, n_8, n_{10}]; the values of QV are all *true*. The list phase is done.

We continue the processor-selection phase by deploying tasks from list L in FIFO manner to suitable processor. According to L = [n_1, n_4, n_2, n_5, n_9, n_3, n_7, n_6, n_8, n_{10}], at the beginning, task n_1 is assigned to processor p_3 because it produces the earliest finish time, i.e., $EFT(n_1) = 9$ and $TP(n_1) = p_3$. Then, n_4 is the next task to be removed from L. The $LMAT(n_4, p_1) = \text{Max}(EFT(n_1) + 9) = 18$, according to the partial schedule, the

$ST(n_4, p_1) = 18$ and $FT(n_4, p_1) = ST(n_4, p_1) + w_{4,1} = 18 + 13 = 31$. We have $FT(n_4, p_2) = 18 + 8 = 26$ and $FT(n_4, p_3) = 9 + 18 = 27$. Therefore, the $EFT(n_4) = \text{Min}\{31, 26, 27\} = 26$ and $TP(n_4)$ is p_2 since p_2 is the best choice among the processors.

1. **Procedure** Critical-task Anticipation:
2. Initially, construct an array of Boolean QV and a stack S.
3. QV[n_j] = *false*, $\forall\, n_j \in$ V.
4. Push n_{exit} on top of S.
5. **While** S is not empty **do**
6. Peek task n_j on the top of S;
7. **If**(all QV[n_i] are *true*, for all $n_i \in pred(n_j)$ or task n_j is n_{entry})
8. Pop task n_j from top of S and put n_j into scheduling-list L;
9. QV[n_j] = *true*;
10. **EndIf**.
11. **Else** /* search the $CT(n_j)$ */
12. **For** each task n_i, where $n_i \in pred(n_j)$ **do**
13. **If**(QV[n_i] = *false*)
14. Put $CS(n_i)$ into container C;
15. **Endif**
16. **EndFor**
17. Push tasks $pred(n_j)$ from C into S by non-decreasing order according to their critical scores;
18 Reset C to empty;
19. /* if there are 2+ tasks with same $CS(n_i)$, task n_i is randomly pushed into S. */
20. **EndElse**
21. EndWhile

Fig. 3. The Critical-Task Anticipation Procedure

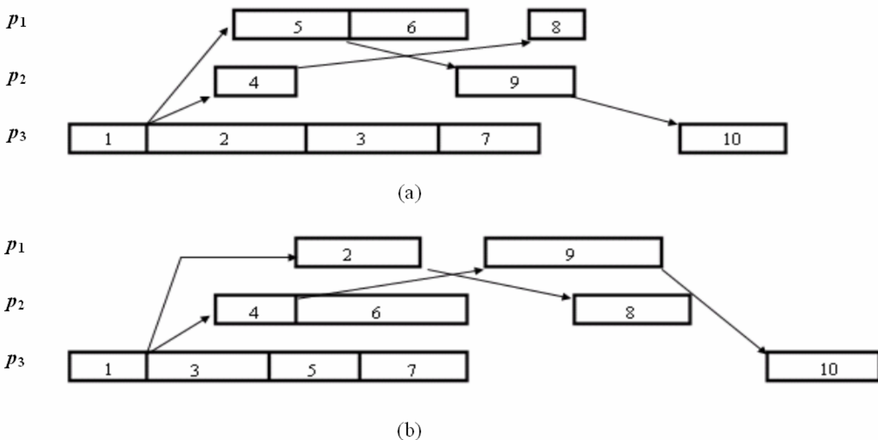


Fig. 4. Scheduling results for the DAG in Fig. 1 using (a) *CA* algorithm (makespan = 81) (b) *HEFT* algorithm (makespan = 92)

The scheduling result obtained by the *CA* algorithm for the DAG given in Fig. 1 is depicted in Fig. 4(a). On the other hand, the *HEFT* results L = [n_1, n_4, n_3, n_2, n_5, n_6, n_9, n_7, n_8, n_{10}] in the list phase and the scheduling result is given in Fig. 4(b) and it demonstrates that the *CA* algorithm outperforms the *HEFT* algorithm in terms of makespan.

For algorithm complexity, the time complexity of the *CA* algorithm for calculating critical score is $O(|P|+|E|)$, where $O(|P|)$ is for $\overline{w_i}$ calculation. The procedure of Critical-task Anticipation leads $O(|E|+|V|)$ time complexity in the list phase and takes $O(|E| \times |P|)$ in the processor-selection phase. Therefore, the time complexity of the *CA* algorithm is $O(|E| \times |P|)$.

5 Simulation

In this section, we first introduce the random graph generator, a simulator that generating weighted directed acyclic graphs with various characteristics. We then explain metrics for performance comparison. Finally, we show the simulation results.

5.1 Random Graph Generator

To evaluate the efficiency of our algorithm, we implemented a Random Graph Generator (RGG) to simulate applications with various characteristics. RGG uses the following input parameters to produce diverse graphs.

- Weight of graph (*weight*), which is a constant = {32, 128, 512, 1024}.
- Number of tasks in the graph (n). In our simulation, n = {20, 40, 60, 80, 100}.
- Parallelism of graph (p)
  It influences the shape of the graph. The p is assigned for 0.5, 1.0 and 2.0. The level of graph is $\lfloor \sqrt{v} / p \rfloor$. For example, if the value p = 2.0, it will generate higher parallelism graph and vice versa.
- Out degree of a task (d).
  The d is assigned for 1, 2, 3, 4 and 5. The out degree represents the dependence among tasks. If the degree is large, the task relationship is high.
- Heterogeneity of computation cost (h).
  This parameter is used to control the computation cost $w_{i,k}$ for a task n_i on processor p_k. The $w_{i,k}$ is randomly chosen from the following formula.

$$w_i \times \left(1 - \frac{h}{2}\right) \leq w_{i,k} \leq w_i \times \left(1 + \frac{h}{2}\right). \tag{7}$$

  RGG randomizes w_i from the interval [1, *weight*]. Note that if the *weight* is assigned with larger value, it represents the estimation of great precision. The h is assigned for 0.1, 0.25, 0.5, 0.75 and 1.0.
- Communication to Computation Ratio (*CCR*).
  The *CCR* is assigned for 0.1, 0.5, 1.0, 2.0 and 10.0.

5.2 Comparison Metrics

As mentioned earlier, the objective of our scheduling algorithm is to shorten the completion time of an application. Several comparative metrics are given below:

- Makespan
  The *makespan* (also known as schedule length) is defined as

$$makespan = \max(EFT(n_i), \ for\ all\ i = 1 \sim n \tag{8}$$

- Speedup
  The speedup is defined as

$$Speedup = \frac{\min_{p_j \in P}\{\sum_{n_i \in V} w_{i,j}\}}{makespan} \tag{9}$$

  The numerator is the minimal accumulated sum of computation cost of tasks which are assigned on one processor. The meaning of Speedup is comparison between sequential execution time and parallel execution time.
- Percentage of Quality of Schedules (PQS)
  The percentage of the *CA* algorithm produces better, equal and worse quality of schedules compared to the *HEFT* algorithm.

5.3 Simulation Results

In [14], *HEFT* demonstrated superior performance to other scheduling techniques, the Dynamic Level Scheduling Algorithm [13], the Levelized-Min Time Algorithm [5], and the Mapping Heuristic Algorithm [12]. Upon this reason, in this simulation, our emphasis is on the performance comparison with *HEFT*. The first simulation aims to demonstrate the merit of the *CA* algorithm by showing the quality of schedules using the RGG. Figures 5 and 6 show the simulations make use of the parameters which generate 1875 different DAGs. The *CA* scheduling algorithm provides superior performance for 70% ~ 80% test samples. Fig. 5 (a) shows the effect of setting different *weight* = {32, 128, 512, 1024}. This result shows that PQS does not changed largely by varying the *weight*. Therefore, it is interesting to discover the effect on different number of processors. Fig. 5 (b) shows that the *CA* algorithm performs very well when the number of processor becomes large.

weight	32	128	512	1024
CA Better:	74.61%	73.22%	72.42%	73.13%
Equal:	0.21%	0.05%	0.05%	0.05%
CA Worse:	25.18%	26.73%	27.53%	26.82%

(a)

Processors	5	6	7	8
Better:	77.41%	80.45%	82.56%	85.46%
Equal:	0.10%	0.00%	0.16%	0.10%
Worse:	22.49%	19.55%	17.28%	14.44%

(b)

Fig. 5. PQS (a) *CA* compared with *HEFT* (3 processors) (b) *CA* compared with *HEFT* (*weight* = 128)

Figures 6 present the simulation results in terms of speedup by varying *n*, *p*, *d*, *CCR* and *h*, respectively. The effect of number of task is shown in Fig. 6 (a). For both algorithms, while the simulation has small number of processors, the speedup is placid. However, when we adapt processors to eight, it is apparent that speedup increased

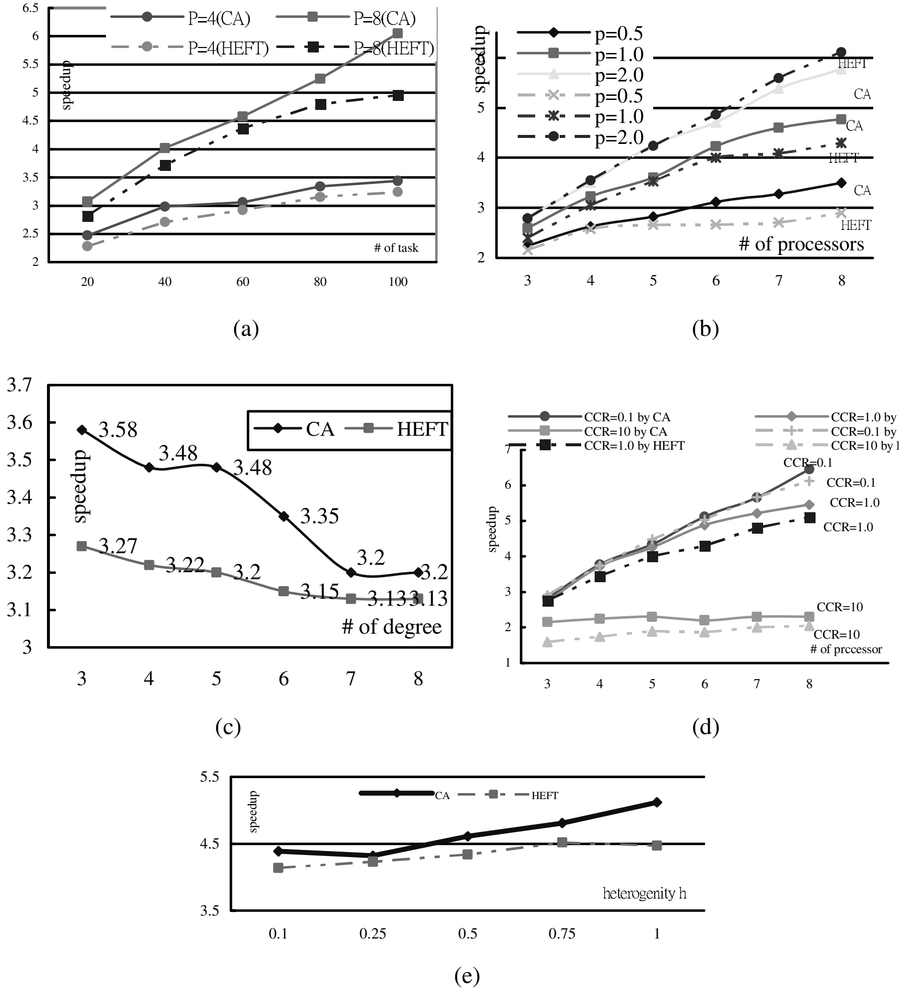


Fig. 6. Performance comparison of the *CA* and the *HEFT* algorithms (a) speedup comparison with different number of tasks (n) (b) speedup comparison with different degree of parallelism (p) (c) speedup comparison with different out-degree of tasks (d) (d) speedup comparison with different *CCR* (e) speedup comparison with different heterogeneity of computation cost (h)

significantly, especially in the situation of large number of task. Compare with the *HEFT* algorithm, the improvement rate of the *CA* algorithm in terms of average speedup is about 7% at processor = 4 and 11% at processor = 8; the Improvement Rate (IR_{CA}) is estimated by the following equation:

$$IR_{CA} = \frac{\sum speedup\ (CA) - \sum speedup\ (HEFT)}{\sum speedup\ (HEFT)} \quad (10)$$

Fig. 6 (b) helps in investigating the sensitivity of task parallelization. It is noticed that, when *p* is large, the graphs are tending parallelism. As shown in Fig. 6 (b), the *CA* algorithm favors linear graphs (*p*=0.5), also outperforms the *HEFT* algorithm in general graphs too (*p*=1.0), but is defeated in high parallelism graphs (*p* =2.0). Fig. 6 (c) gives the observation about the dependence relationship among tasks by fixing number of processors at 5. Although the speedups of both algorithms are stable, the *CA* algorithm outperforms the *HEFT* in most cases. In Fig. 6(d), the impact of communication on speedup is plotted for various value of *CCR*. We vary *CCR* by 0.1, 1.0 and 10. It is noted that an increase in *CCR* decreases the speedup rapidly. For example, speedup offered by the *CCR*=0.1 used *CA* at processor = 8 is 6.45 and *CCR* =10.0 used *CA* at processor =8 is only 2.2. This is due to the fact that when the communication is higher than computation, the behavior of migration of tasks is not useful. Beside, when the *CCR* is large, there is still poor performance even if the numbers of processors are added. Namely, there is no benefit of increase of processors when communication is the bottleneck. Fig. 6 (e) shows the effect of heterogeneity (*h*) by fixed number of processor =8. From Fig. 6 (e), we observe that the speedup increases with increasing *h* in both algorithms. As the result of simulation, we consider the *CA* algorithm achieves significant performance improvement in majority part.

6 Conclusion

In this paper, we proposed a new scheduling heuristic, the critical-task anticipation (*CA*) algorithm for heterogeneous computing systems. The *CA* scheduling algorithm is a list scheduling heuristic and has a simple structure and low complexity.

For performance evaluation, we compared *CA* with *HEFT* scheduling algorithm. The experimental results showed that *CA* is in most cases equal or superior to *HEFT* due to a more appropriate task prioritizing. Graphs with medium and high *CCR* were always best scheduled by *CA*. In the case of low *CCR*, the *CA* algorithm delivered comparable results to the *HEFT* algorithm. Overall speaking, from the simulation, the performance of the *CA* algorithm has been observed to fit most *DAG*.

References

1. R. Bajaj and D. P. Agrawal, “Improving Scheduling of Tasks in a Heterogeneous Environment,” *IEEE Transactions on Parallel and Distributed Systems,* vol. 15, no. 2, pp. 107-118, 2004.
2. S. Behrooz, M. Wang, and G. Pathak, “Analysis and Evaluation of Heuristic Methods for Static Task Scheduling,” *Journal Parallel and Distributed Computing,* vol. 10, pp. 222-232, 1990.
3. A. Gerasoulis and T. Yang,” On the Granularity and Clustering of Directed Acyclic Task Graphs,” *IEEE Transactions on Parallel and Distributed Systems,* vol.4, no.6, pp. 686-701, 1993.
4. T. Hagras and J. Janecek, ” A High Performance, Low Complexity Algorithm for Compile-Time Task Scheduling in Heterogeneous Systems,” *IEEE Proc. IPDPS*, 2004.

5. M. Iverson, F. Ozguner, and G. Follen, "Parallelizing Existing Applications in a Distributed Heterogeneous Environment," *Proc. Heterogeneous Computing Workshop,* pp. 93-100, 1995.
6. Y. Kwok and I. Ahmed, "Benchmarking the Task Graph Scheduling Algorithms," *Proc. IPPS/SPDP,* 1998.
7. J. Liou and M. A. Palis, "A Comparison of General Approaches to Multiprocessor Scheduling," *Proc. Int'l. Parallel Processing Symposium*, pp. 152-156, 1997.
8. S. S. Pande, D. P. Agrawal, and J. Mauney, "A Scalable Scheduling Method for Functional Parallelism on Distributed Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems,* vol. 6, no. 4, pp. 388-399, 1995.
9. C.I. Park and T.Y. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication," *IEEE Transactions on Computers*, vol. 51, no. 4, pp. 444-448, 2002.
10. A. Radulescu and A. van Gemund, "Fast and effective task scheduling in heterogeneous systems," *Heterogeneous Computing Workshop*, *2000*, pp. 229-238, May, 2000.
11. S. Ranaweera and D. P. Agrawal, "A Task Duplication Based Scheduling Algorithm for Heterogeneous Systems," *IEEE Proceedings of IPDPS*, pp. 445-450, 2000.
12. H. 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.
13. G. C. Sih and E. A. Lee, "A Compile Time Scheduling Heuristic for Interconnection - Constrained Heterogeneous Processors Architectures," *IEEE Transactions on Parallel and Distributed Systems,* vol. 4, no. 2, pp. 175-187, 1992.
14. H. Topcuoglu, S. Hariri, and W. Min-You, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Transactions on Parallel and Distributed Systems,* vol.13, no. 3, pp. 260-274, 2002.
15. M. Wu and D. Gajski, "Hypertool: A Programming Aid for Message-Passing System," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 3, pp.330-343, 1990.
16. T. Yang and A. Gerasoulis, "DSC:Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Tran. on Parallel and Distributed Systems,* vol. 5, no.9, pp. 951-967, 1994.