Simulation Modelling Practice and Theory 63 (2016) 83–103

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

A fast predictive algorithm with idle reduction for heterogeneous system scheduling

Anan Niyom[*], Peraphon Sophatsathit, Chidchanok Lursinsap

Advanced Virtual and Intelligent Computing Research Center, Department of Mathematics and Computer Science, Faculty of Science, Chulalongkorn University, Thailand

A R T I C L E I N F O

Article history:
Received 30 September 2015
Revised 9 February 2016
Accepted 9 February 2016
Available online 9 March 2016

Keywords:
Task graph
Heterogeneous systems
Algorithm running time
Schedule length

A B S T R A C T

A heterogeneous task scheduling algorithm called Predict and Arrange Task Scheduling (PATS) algorithm was proposed to achieve a lower bound time complexity with minimum schedule length. Two major steps were introduced, i.e. earliest finish time with level-based task scheduling and idle slot reduction. In the first step, tasks are scheduled according to their predicted earliest finish time from the candidate task list and their dependencies. Scheduling is performed one level at a time starting from top level and transcend downward. In the second step, the idle time slots in each processing unit are minimized. Two sets of experiments were designed to evaluate the merits of proposed algorithm. The first experiment involved the task graphs used by other methods. These graphs are all synthesized. The second experiment concerned the task graphs derived from real world applications such as montage work flow, molecular dynamic code. The experimental results showed that the PATS algorithm yielded better average schedule length ratio, running time, and efficiency than the compared algorithms.

1. Introduction

A heterogeneous system is more practical than a homogeneous system for distributed computing due to the actual availability of various processing units with different computing capabilities. The processing speed of each unit and also the installed software applications may be different. Scheduling and assigning a set of tasks from a host unit to proper heterogeneous units to achieve a minimum response time as well as minimum energy consumption is very challenging. Usually, a set of tasks is represented by a directed acyclic task graph. Recently proposed algorithms for scheduling and assigning tasks in heterogeneous units focused on two optimization issues concerning energy conservation and makespan reduction. The first issue of energy conservation covers the minimization of energy usage of the system and at the same time the process to shorten the scheduled makespan under limited power [1–6]. The second issue of makespan reduction emphasizes the process to reduce the scheduled time span or idle slots. Typically, minimizing energy consumption of a system employs basic methods such as server selection, task ordering, and simulation technologies such as dynamic voltage scaling, dynamic frequency scaling, and dynamic power management [7–12]. But reducing makespan must involve the minimization of schedule length in various situations [13,14]. The underlying algorithms for these two issues can be categorized into four groups which are list-based, clustering-based, duplicate-based, and genetic based algorithms.

[*] Corresponding author. Tel.: +66 877127360.
E-mail addresses: anan.niyom@gmail.com (A. Niyom), speraphon@gmail.com (P. Sophatsathit), lchidcha@gmail.com (C. Lursinsap).

http://dx.doi.org/10.1016/j.simpat.2016.02.006
S1569-190X(15)30094-0/© 2016 Elsevier B.V. All rights reserved.

A list-based scheduling separates task prioritization and server selection [15,16] by traversing throughout a dependent task graph to order the tasks for scheduled assignment on suitable processing units. The list of ordering is an important output which determines the performance of the algorithm. A duplicate-based scheduling is similar to that of the list-based scheduling but redundant tasks are added on different processing units to reduce communication cost and the consequent schedule length [17,18]. A clustering-based scheduling groups tasks suitable for each processing unit and orders the tasks in each cluster [19,20]. It also reduces the communication or computation costs. A genetic algorithm [21,22] is based on natural and genetic selection to find an optimum solution to the problem. Moreover, the first three groups are classified as heuristic whereas the last group is considered an evolutionary algorithm. An evolutionary algorithm usually takes longer time to reach the desired solution although the scheduling length is minimized at the expense of extra time. Its running time could vary greatly for problems with the same number of input tasks and processing units depending on the solution refinement process to assign input tasks on a suitable processing unit. On the other hand, a heuristic algorithm usually focuses on finding the solution using the least extra scheduling overhead. However, the yielded scheduling length so obtained is not always the shortest. Therefore, both evolutionary and heuristic algorithms are appropriated in different situations. For example, in the situation where the same task graph is to be processed many times, the solution obtained from an evolutionary algorithm which yields the shortest scheduling length is preferable since the extra scheduling overhead is only necessary during the initial processing of the task graph. The heuristic approach is, on the other hand, favorable in the situation where varieties of task graphs are to be processed.

In this study, we developed the heuristic scheduling algorithm with emphasis on the problem of scheduling a task graph, whose tasks have different computational requirements, in a heterogeneous computing system. All processing units in the system have different computing speeds and different available software applications. The objectives are to minimize the length of makespan of the given task graph and time complexity of scheduling process. Since the previously imposed constraints of the list-based scheduling approach is similar to ours, therefore the concept of list-based scheduling was adopted and modified to solve our studied problem.

The rest of this paper is organized as follows. Section 2 summarizes the concept of each compared algorithm. Section 3 formulates the studied problem and constraints. Section 4 explains the proposed algorithm. Sections 5 provides the experimental procedures and results. Section 6 discusses the experimental results. Section 7 concludes the paper.

2. Related works

Many algorithms for task assignment in heterogeneous systems have been proposed. We will briefly summarize those algorithms that are pertinent to our study, such as heterogeneous earliest finish time, low complexity performance effective task scheduling, look-ahead, constrained earliest finish time, and predict earliest finish time algorithm.

Heterogeneous earliest finish time or HEFT algorithm was proposed by Topcuoglu et al. [23]. This algorithm consists of two phases: task prioritizing and processor selection. The task prioritizing phase calculates the priority queue in an upward ranking called $rank_u$ based on computation and communication costs for each task. The processor selection phase calculates the earliest execution finish time using insertion-based policy to assign any pending tasks in an idle time slot lying between already scheduled tasks on a processing unit. This algorithm has $O(v^2 \times p)$ time complexity for v vertices or number of tasks and p processing units.

Low complexity performance effective task scheduling (PETS) algorithm was proposed by Ilavarasan and Thambidurai [24]. This algorithm constructs a priority queue by sorting each task on each scheduling level according to average computation cost, data transfer cost, and rank of predecessor tasks. The algorithm computes the earliest finish time of each task and assigned a task by insertion-based policy in an idle time slot between already scheduled tasks on a processing unit. The time complexity of this algorithm is equal to $O(v^2)(p \times log\, v)$ for v vertices or number of tasks and p processing units.

Lookahead algorithm was proposed by Bittencourt et al. [25]. This algorithm, based on HEFT algorithm [23], calculates the estimated finish time of all children of the considered task in queue of $rank_u$ to find the shortest finish time of subsequent tasks. Therefore, it increases the time complexity of the original HEFT by a factor of $(p \times c)$ for p processing units and average c children per task. In case of tasks spreading evenly on each level of scheduling graph, the average number of children must not exceed v/l, where l is the number of levels in the graph. The time complexity of lookahead HEFT algorithm is equal to $O(v^3 \times p^2/l)$.

Constrained earliest finish time (CEFT) algorithm was proposed by Khan [26]. This algorithm find a set of constrained critical paths by traveling and pruning downward from the start task to the exit task of the graph. A constrained critical path is defined in terms of average execution cost, transmission weight, and weight of predecessor of each task. A schedule queue is established by selecting a task or more than one task ready to be scheduled from the longest constrained critical path. The process is repeated for the second longest constrained critical path, and so on, until all tasks are added to the schedule queue in a round-robin manner. The obtained schedule queue contains several constrained critical sub-paths, each of which may have more than one task. These constrained critical sub-paths are then assigned to the processing unit to yield the minimum finish time. The time complexity of this algorithm is equal to $O(n \times p \times (n + m + deg_{in}))$ for v vertices or number of tasks, p number of processing units, m edges, and deg_{in} in-degrees of task graph.

Predict earliest finish time (PEFT) was proposed by Arabnejad and Barbosa [27]. This algorithm is based on optimistic cost table. There are two phases which are computing task prioritization and processor selection. Task prioritization is determined from sorted average of optimistic cost tables on each processor. A processor is selected from the results of task optimistic earliest finish time (O_{EFT}) determined from the minimal summation of optimistic cost tables and earliest finish time. The tasks are then scheduled to each processing unit using insertion-based policy. The time complexity of this algorithm is in the order of $O(v^2 \times p)$ for v tasks and p processing units.

Most proposed algorithms use all tasks in the DAG for generating the priority queue and finding ready tasks from the graph to assign. This is one cause of increasing running time complexity of the algorithms. The level-based approach turns out to be a better way to reduce running time. Hence, a heterogeneous scheduling algorithm which yields good performance and time complexity in various types of task graphs was proposed in this paper. The proposed algorithm is tested on performance against the previously mentioned algorithms by using a wide range of dependent task scenarios having different structures, communication to computation ratio, number of tasks, and processing units. The details on the proposed algorithm will be given in the following sections.

3. Problem formulation and definitions

Let $G = (\mathbf{V}, \mathbf{E})$ be a dependent task graph consisting of sets of tasks $\mathbf{V} = \{v_1, \ldots, v_n\}$ and edges $\mathbf{E} = \{(v_i, v_k) \mid v_i, v_k \in \mathbf{V}\}$. An edge (v_i, v_k) means that task v_i is performed prior to task v_k. Suppose the dependent task graph G is at a host unit. Given a set of processing units $\{p_1, \ldots, p_n\}$ with different c omputing capabilities, we would like to cluster all tasks $v_i \in \mathbf{V}$ into appropriate task group and assign each task group to a service unit p_a capable of executing all tasks in the group so that the minimum makespan can be achieved. Furthermore, the time spent to cluster and assign these task groups must also be minimum. Each task v_i requires a specific computing capability and also specific application software. Note that, for any task v_i, it is possible that there may be more than processing units capable of executing v_i. Let $\mathbf{P}_{v_i}$ denote the only set of processing units capable of executing task v_i. Obviously, each processing unit in $\mathbf{P}_{v_i}$ spends different amount of computing time to execute v_i. The relevant terminologies are defined in the following sections.

3.1. Relevant scheduling time points

There are several important points of time to be considered in a scheduling process. In this study, four relevant points of time were focused, i.e. *actual finish time, available time* of processing unit (or available_time), *earliest start time*, and *earliest finish time*. The detail of each term is described below.

Definition 1. *Actual finish time* of task v_j, denoted as $AFT(v_j)$, is the point of time where the execution of task v_j is finished by some assigned processing unit. Note that this is not the estimated finishing time determined during the scheduling process.

Definition 2. *Available time* of an available processing unit p_a, denoted as $available\_time(p_a)$, is the point of time where processing unit p_a is neither executing any task nor waiting for the resulting data of predecessor tasks from other processing units.

Definition 3. *Earliest start time* of task v_i at processing unit p_a, denoted as EST_{v_i,p_a}, is the earliest point of time to start the execution of task v_i at assigned processing unit p_a.

The value of EST_{v_i,p_a} depends upon the following scheme. As soon as processing unit p_a receives the resulting data from v_j and it is still in the idle state, processing unit p_a can immediately start executing v_i. If processing unit p_a is not in the idle state, then task v_i must wait until processing unit p_a is available. Let $\mathcal{A}_{v_i}$ be the set of all predecessor tasks of v_i. Therefore, the value of EST_{v_i,p_a} is equal to

$$EST_{v_i,p_a} = \max\left(\max_{v_j \in \mathcal{A}_{v_i}}\left(AFT\left(v_j\right) + w_{p_a}(v_i)\right),\ \ available\_time(p_a)\right) \tag{1}$$

where $w_{p_a}(v_j)$ is the amount of time to transmit the resulting data of task v_j after executing on other processing unit to processing unit p_a. If a descendant task v_i which is waiting for the resulting data from predecessor task v_j is on the same processing unit as v_j, then the communication cost is assumed to be zero.

Definition 4. *Earliest finish time* of task v_i executed by processing unit p_a, denoted as EFT_{v_i,p_a}, is the point of time where the execution of task v_i is finished.

There are two possible scenarios concerning this value. The first scenario occurs when task v_i starts its execution at EST_{v_i,p_a}. The starting of v_i must not change the starting of any assigned task v_k in the same processing unit p_a. The value of EFT_{v_i,p_a} is equal to

$$EFT_{v_i,p_a} = EST_{v_i,p_a} + t_{p_a}(v_i), \tag{2}$$

where $t_{p_a}(v_i)$ is the amount of time used to execute task v_i on processing unit p_a. The second scenario occurs when the length of available time slot is shorter than the actual execution time of v_i. This implies that any assigned tasks v_k in the same processing unit p_a must be shifted their starting times to new starting times. Let $\mathcal{V}$ be the set of all shifted tasks. The value of EFT_{v_i,p_a} in this scenario becomes

$$EFT_{v_i,p_a} = EST_{v_i,p_a} + t_{p_a}(v_i) + \sum_{v_k \in \mathcal{V}} t_{p_a}(v_k) \tag{3}$$

3.2. Comparison metrics

The performance of PATS algorithm and other comparatvie algorithms is evaluated by following metrics:

- *Schedule length or makespan*. The schedule length or *makespan* denotes the point of time where execution of the latest task v_f finishes. It is the summation of $AFT(v_f)$ from all processing units. This schedule length is the output of all task schedule.
- *Average schedule length ratio*. The schedule length ratio or *SLR* is another metric used to compare scheduling algorithms in real world application graphs. This metric is defined by Amoura et al. [13] and is commonly used to evaluate a scheduling algorithm. The SLR value is computed by the ratio of makespan and summation of the minimum calculation time of all tasks on the critical path (CP).

$$SLR = \frac{makespan}{\sum_{v_i \in CP} \min_{p_a \in \mathbf{P}_{v_i}} (t_{p_a}(v_i))} \tag{4}$$

- *Efficiency*. This is another performance evaluation metric defined as the ratio between the lower bound of schedule length and the makespan. The lower bound is defined as the summation of the minimum time of all tasks divided by the number of processing units used. The closer is the efficiency value to unity, the higher is the performance.

$$Efficiency = \frac{LowerBound}{makespan} \tag{5}$$

$$LowerBound = \frac{\sum_{v_i \in \mathbf{V}} \min_{p_a \in \mathbf{P}_{v_i}} (t_{p_a}(v_i))}{p} \tag{6}$$

- *Algorithmic running time*. The algorithmic running time (*ART*) of an algorithm is the amount of time spent by the scheduling algorithm. It could be viewed as an overhead of determining the schedule length.
- *System finish time*. The system finish time (*SFT*) metric combines the effects of the schedule length and the algorithm running time metrics. This metric takes into account an overhead of time requires by the algorithm to determine the solution with a certain scheduling length. The metric is calculated by using Eq. (7).

$$SFT = makespan + ART \tag{7}$$

- *Performance percentage*. This metric is used to evaluate the system finish time in terms of its percentage of a reference heuristic scheduling algorithm. The metric is defined as the ratio between the system finish time of the algorithm and that of the reference algorithm, Eq. (8). For this paper, the HEFT algorithm [23] which is considered a de facto standard for heuristic scheduling algorithms yields reasonably short scheduling length with low running time for various types of task graphs [24–27]. Moreover, the solution of HEFT algorithm is in second order of polynomial time which is low in time complexity.

$$Performance\ Percentage = \frac{SFT_{ref}}{SFT} \times 100 \tag{8}$$

4. Proposed algorithm

The proposed *Predict and Arrange Task Scheduling* (PATS) algorithm consists of two phases. In the first phase, Algorithm 1 arranges all the tasks to attain the shortest schedule length. In the second phase, Algorithm 2 reduces the idle slots produced from the first phase to shorten the schedule length. The brief explanation of each phase is as follows.

Phase 1

For each task v_i, the prime candidate processing unit p_a from set $\mathbf{P}_{v_i}$ is determined from the estimated earliest start time EST_{v_i,p_a} and the earliest finish time EFT_{v_i,p_a}. On any level l of the dependent task graph G, an earliest-finish-time candidate task list is built to predict which task has the shortest finish time. The first task in the list is taken to be the first candidate to determine the available time slot in a processing unit. If the determined time slot is large enough for the task to fit in, then the task can directly be assigned to this time slot. But if it is too small for the consider task, the determined time slot must be extended by shifting other tasks already assigned to some other starting time points so that the considered task can be put into the time slot. The processing unit executing this first candidate task is marked and assigned as the primary candidate processing unit by insertion method. The remaining tasks on the same level are updated and queued until all the tasks on the same level are assigned. This process repeats for all other remaining levels of the graph. The procedural details are given in Algorithm 1.

Algorithm 1 Level-based task assignment by using EFT candidate task list.

```
Require: $v_i$, $\mathbf{P}_{v_i}$, and $G$.
  for each level $l$ of graph $G$ do
    Let $\Phi_l$ be a set of tasks $v_i$ on level $l$.
    Find task $v_r \in \Phi_l$ that can be executed by only one processing unit.
    Assign $v_r \in \Phi_l$ to the earliest time that its processing unit is ready to execute.
    Compute $EST_{v_i,p_a}$ of each $v_i \in \Phi_l$ by using Eq. (1).
    while $\exists$ unassigned $v_i$ in $l$ do
      for all tasks $v_i \in \Phi_l$ do                                        ▷ build EFT candidate task list
        for each processing unit $p_a \in \mathbf{P}_{v_i}$ do
          $EST_{v_i,p_a} = \max\left(EST_{v_i,p_a}, available\_time(p_a)\right)$          ▷ update $EST_{v_i,p_a}$
          if assigning $v_i$ at $EST_{v_i,p_a}$ does not shift any already scheduled task then
            $EFT_{v_i,p_a} = EST_{v_i,p_a} + t_{p_a}(v_i)$
          else
            Let $\mathcal{V}$ be the set of all shifted tasks.
            $EFT_{v_i,p_a} = EST_{v_i,p_a} + t_{p_a}(v_i) + \sum_{v_j \in \mathcal{V}} t_{p_a}(v_j)$
          end if
          Mark $v_i$ as $v_k$ if it has the furthest EFT.
        end for
      end for
      Let $p_a$ be the processing unit that gives the shortest EFT for $v_k$.
      Mark $p_a$ as the primary candidate processing unit.
      Assign $v_k$ to the first order of candidate processing unit in $\mathbf{P}_{v_k}$ at time slot $EST_{v_k,p_a}$.
      Remove $v_k$ from $\Phi_l$.
      Mark predecessor task $u_{v_k}$ as the latest finish receiving data of task $v_k$.
    end while
  end for
  return The scheduled list of tasks $v_i$ on its assigned processing unit.
```

Algorithm 2 Reducing idle slot in task scheduling.

```
Require: $v_i$, $G$, $u_{v_k}$, and scheduled list of tasks $v_i$.
 1: Let $v_f$ be the latest task to finish.
 2: Let $v_s = v_f$.
 3: while task $v_s$ is not in the first level do
 4:   Traverse the task list upward starting at $v_s$ until an empty slot is found.
 5:   Let $v_k$ be the task before this empty slot.
 6:   Let $u_{v_k}$, which is the latest finish receiving data of task $v_k$, be the marked predecessor of $v_k$.
 7:   if $u_{v_k}$ is possibly executed by the same processing unit, say unit $p_a$, as of $v_k$ then
 8:     Let $p_b$ be the currently assigned processing unit of task $u_{v_k}$.
 9:     Temporarily remove $u_{v_k}$ from its present slot and insert it at the EST of unit $p_a$.
10:     Let $AFT'(v_f)$ be the new actual finish time of task $v_f$.
11:     if $AFT'(v_f) < AFT(v_f)$ then
12:       Permanently assign $u_{v_k}$ to processing unit $p_a$.
13:     else
14:       Remove $u_{v_k}$ from processing unit $p_a$ and reassign task $u_{v_k}$ back to unit $p_b$.
15:     end if
16:   end if
17:   Set task $u_{v_k}$ as the next considered task $v_s$.
18: end while
19: return The scheduled list of all tasks in each assigned processing unit.
```

Phase 2

Schedule length of any processing unit that contains idle slots is further reduced. Since some tasks can be moved or inserted to be executed in the idle slot preceding the task under consideration (current task), the immediate parent of the current task which causes a wait or delay is reassigned to the same processing unit. The procedural details are given in Algorithm 2.

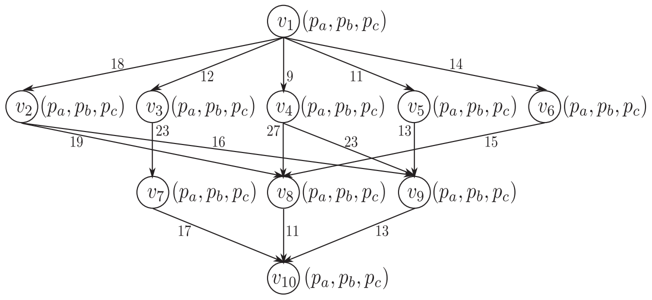

Fig. 1. A dependent task graph using similar transmission and execution costs (the first experiment). The graph was taken from Topcuoglu et al. [23].

Table 1
A list of execution time $t_{p_a}(v_i)$ of each processing unit for dependent task graph of Fig. 1 (the first experiment).

Task	Execution time		
v_i	$t_{p_a}(v_i)$	$t_{p_b}(v_i)$	$t_{p_c}(v_i)$
1	14	16	9
2	13	19	18
3	11	13	19
4	13	8	17
5	12	13	10
6	13	16	9
7	7	15	11
8	5	11	14
9	18	12	20
10	21	7	16

To illustrate how these two algorithms work, we will use the dependent task graph of Fig. 1 as an illustration, along with possible execution time of each task by each processing unit as shown in Table 1. The weight on each edge denotes the transmission time. Each vertex is labeled with task name and executable processing units described in parenthesis. For example, vertex v_1 will take 14, 16, 9 units time to be executed on processing unit p_a, p_b, p_c, respectively.

Scheduling is performed first by Algorithm 1. The first level of the graph has only one unassigned task v_1. Task v_1 is tentatively assigned to processing unit p_c having the shortest $EFT_{v_1,p_c} = 9$ units time and processing unit p_c is marked as the primary processing unit of task v_1. On the second level, the EFT of each unassigned task is used to select the proper processing unit. There are five tasks which are v_2, v_3, v_4, v_5 , and v_6. Consider task v_2 being executed on processing unit p_b that gives the longest $EFT_{v_2,p_b} = 46$. Alternatively, the shortest EFT can be achieved when it is assigned to processing unit p_c, that is, $EFT_{v_2,p_c} = 27$. Thus, task v_2 is assigned to processing unit p_c which is marked as the primary processing unit of task v_2. Next, the EFT for other unassigned tasks are recalculated without the inclusion of task v_2 and the longest $EFT_{v_3,p_c} = 46$, and the shortest $EFT_{v_3,p_a} = 32$. Task v_3 is then assigned to processing unit p_a which is marked as the primary processing unit of task v_3. Similarly, task v_6 has the longest $EFT_{v_6,p_a} = 45$, and the shortest $EFT_{v_6,p_c} = 36$. Task v_6 is then assigned to processing unit p_c which is marked as the primary processing unit of task v_6. Continue in this manner, task v_4 has the longest $EFT_{v_4,p_c} = 44$, and the shortest $EFT_{v_4,p_b} = 26$. It is assigned to p_b. Task v_5 has the shortest $EFT_{v_5,p_b} = 39$, and is assigned to p_b.

There are three tasks in the third level, namely, v_7, v_8, and v_9. Computations of EFT values yield task v_9 having the longest $EFT_{v_9,p_c} = 72$, the shortest $EFT_{v_9,p_b} = 55$. It is assigned to p_b. Similarly, task v_7 has the longest $EFT_{v_7,p_b} = 70$, the shortest $EFT_{v_7,p_a} = 39$, and is assigned to p_a. Task v_8 has the shortest $EFT_{v_8,p_a} = 58$ and is assigned to p_a. But v_{10} has the shortest $EFT_{v_{10},p_b} = 76$ and is assigned to p_b. The result of primarily assignment obtained from Algorithm 1 is shown in Fig. 2.

Algorithm 2 is applied to reduce schedule length. The algorithm starts from the final finished task. From Fig. 2, processing unit p_b has the final finished task v_{10}. Traversing task v_{10} upward, an empty slot is found after v_9. But the latest task to finish receiving data of task v_{10} is v_8. Thus, v_8 is temporarily inserted within its EST in the time slot of processing unit p_b. In so doing, the schedule length is reduced to 73 units. Thus, v_8 is permanently assigned to processing unit p_b. Continue traversing upward from task v_8, no schedule length reduction can be done. The results after applying Algorithm 2 are shown in Fig. 3.

5. Experimental results

To evaluate the merit of our algorithm, the experimental results were evaluated in two aspects. The first one is the schedule length and the other is the time factors which are the running time of proposed algorithm, average schedule length ratio (SLR), and efficiency and performance percentage. Two sets of dependent task graphs were used to evaluate the performance

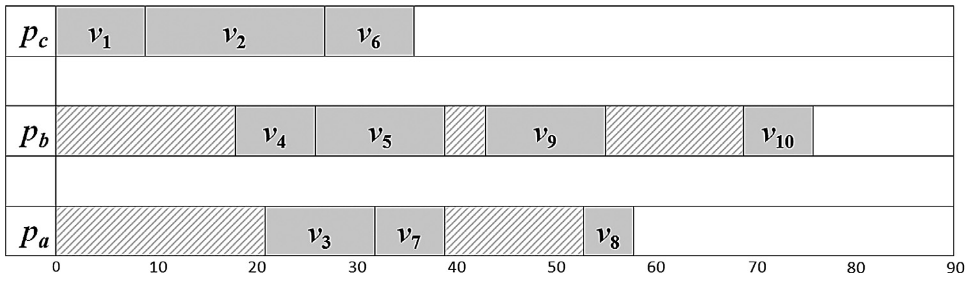

Fig. 2. Results obtained by applying Algorithm 1 of PATS to the dependent task graph of Fig. 1. The dark shaded areas denote an execution state and lighter ones denote an idle state.

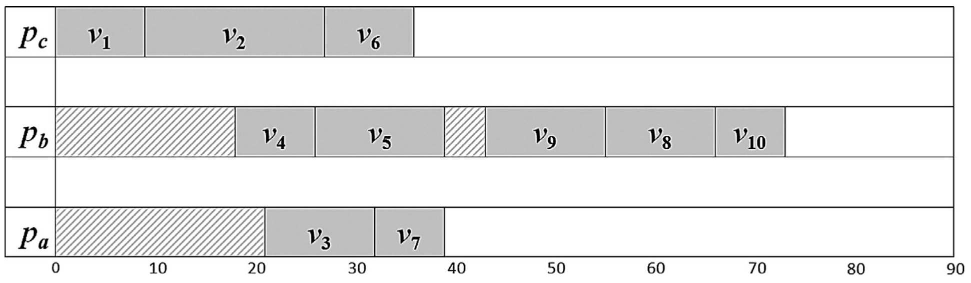

Fig. 3. Results obtained by applying Algorithm 2 of PATS to the dependent task graph of Fig. 1. The dark shaded areas denote an execution state and lighter ones denote an idle state.

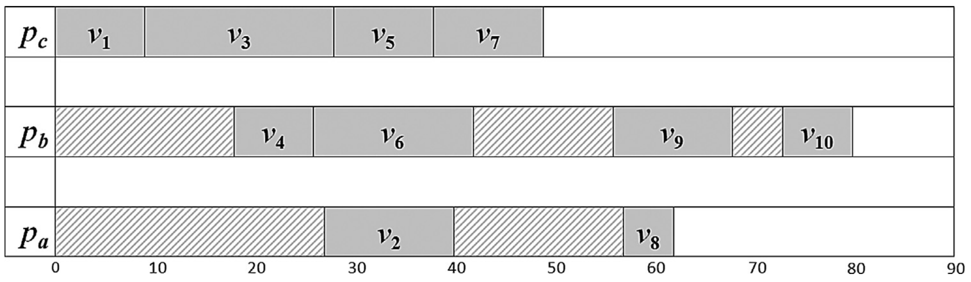

Fig. 4. Result of HEFT algorithm on the dependent task graph of Fig. 1 (the first experiment).

relevant to each aspect. The first set contains all task graphs from other compared algorithms and the second set consists of some dependent task graphs from real world applications which are Molecular Dynamic Code, Montage Work Flow, Gaussian Elimination, and Fast Fourier Transformation with randomly generated computation and communication weights. The detail of results of each considered aspect is the following.

5.1. Dependent task graphs from other compared algorithms

Six tasks graphs from HEFT problem set-up [23], PETS problem set up [24], Lookahead problem set up [25], CEFT problem set up [26], PEFT problem set up [27], and Niyom et al. [6] were used as the test cases. The comparison results of each case is the following.

Test case 1: HEFT problem set-up [23]
The dependent task graph and its data transfer delay $w_{p_a}(v_i)$ are shown in Fig. 1 and the corresponding execution time $t_{p_a}(v_i)$ are shown in Table 1. Based on the structure of this dependent task graph, the number of vertices (v), service processing units (p), and levels (l) are 10, 3, and 4, respectively. The comparison results of the first experiment from compared algorithms are shown in Figs. 4–8. The schedule length of HEFT, PETS, Lookahead, CEFT, PEFT, and PATS algorithms are 80, 76, 76, 81, 85, and 73, respectively. Therefore, PATS algorithm produced the shortest schedule length.

Test case 2: PETS problem set up [24]
The dependent task graph is shown in Fig. 9 and the corresponding execution time $t_{p_a}(v_i)$ are shown in Table 2. By the same scenario as that of test case 1, we set $v = 11,\ p = 3,$ and $l = 5$. The results of schedule length for HEFT, PETS,

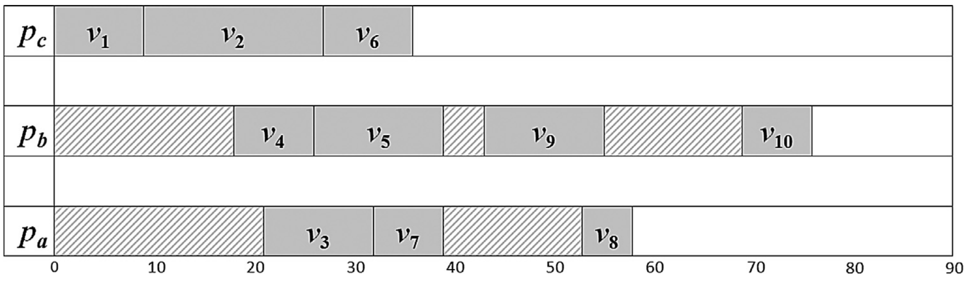

Fig. 5. Result of PETS algorithm on the dependent task graph of Fig. 1 (test case 1).

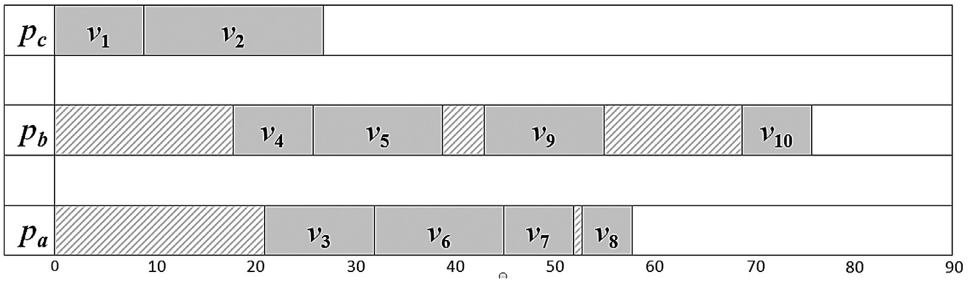

Fig. 6. Result of Lookahead algorithm on the dependent task graph of Fig. 1 (test case 1).

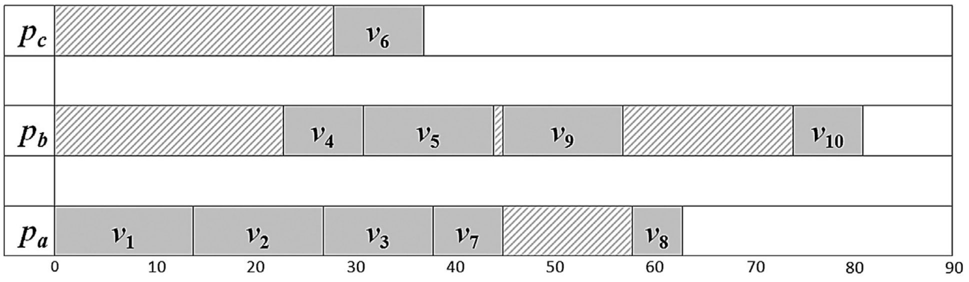

Fig. 7. Result of CEFT algorithm on the dependent task graph of Fig. 1 (test case 1).

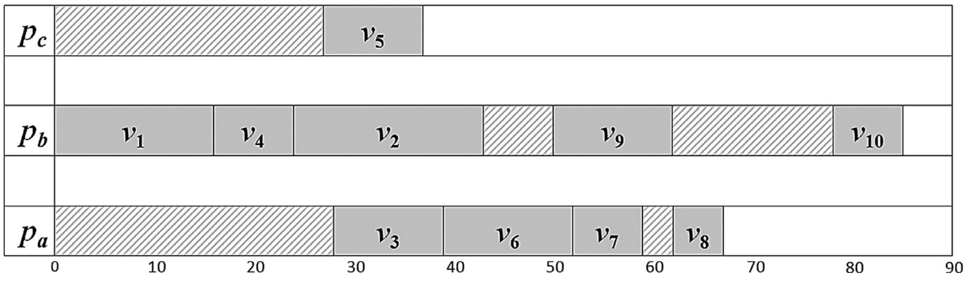

Fig. 8. Result of PEFT algorithm on the dependent task graph of Fig. 1 (test case 1).

Lookahead, CEFT, PEFT, and PATS algorithms are 29, 27, 31, 32, 27, and 28, respectively. Apparently, PETS and PEFT algorithms yielded the shortest schedule length. The schedule length of PATS algorithm is only one unit longer.

Test case 3: Lookahead problem set up [25]

The dependent task graph taken from the Lookahead problem set up [25] is shown in Fig. 10 and the corresponding execution time $t_{p_a}(v_i)$ for each p_i and v_j are shown in Table 3. We set $v = 9,\ p = 3,$ and $l = 4$. The results of schedule length for HEFT, PETS, Lookahead, CEFT, PEFT, and PATS algorithms are 260, 257, 224, 184, 184, and 184, respectively. Obviously, Lookahead, PEFT, and PATS algorithms yielded the same shortest schedule length.

Test case 4: CEFT problem set up [26]

The dependent task graph taken from the CEFT problem set up [26] is shown in Fig. 11 and the corresponding execution time $t_{p_a}(v_i)$ for each p_i and v_j are shown in Table 4. We set $v = 10,\ p = 3,$ and $l = 5$. The results of schedule

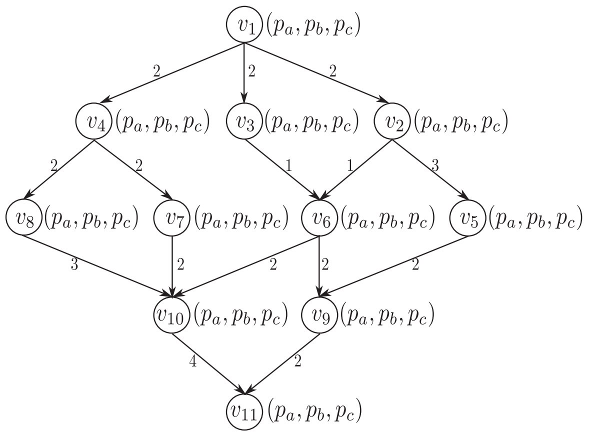

Fig. 9. A dependent task graph taken from Ilavarasan and Thambidurai [24] with its transmission and execution costs (test case 2).

Table 2
Execution time $t_{p_a}(v_i)$ of the task performed on each processing unit (test case 2).

Task	Execution time		
v_i	$t_{p_a}(v_i)$	$t_{p_b}(v_i)$	$t_{p_c}(v_i)$
1	4	4	4
2	5	5	5
3	4	6	4
4	3	3	3
5	3	5	3
6	3	7	2
7	5	8	5
8	2	4	5
9	5	6	7
10	3	7	5
11	5	6	7

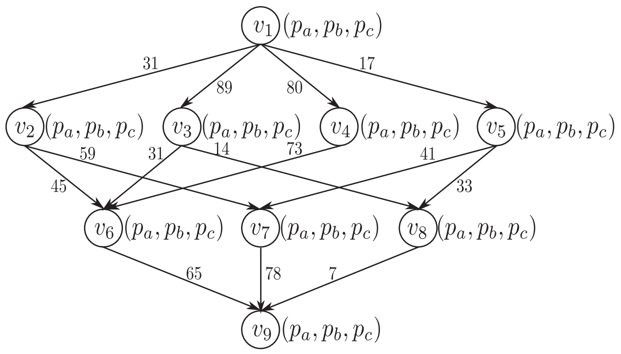

Fig. 10. The dependent task graph using similar transmission and execution costs from [25] (test case 3).

length for HEFT, PETS, Lookahead, CEFT, PEFT, and PATS algorithms are 85, 73, 76, 73, 73, and 73, respectively. The PETS, CEFT, PEFT, and PATS algorithms yielded the same shortest schedule length.

Test case 5: PEFT problem set up [27]

The dependent task graph taken from the PEFT problem set up [27] is shown in Fig. 12 and the corresponding execution time $t_{p_a}(v_i)$ for each p_i and v_j are shown in Table 5. We set $v = 10$, $p = 3$, and $l = 4$. The results of schedule length for HEFT, PETS, Lookahead, CEFT, PEFT, and PATS algorithms are 133, 147, 127, 136, 122, and 124, respectively. Note that PEFT yielded the shortest schedule length but PATS was just two units longer.

Test case 6: Niyom et al. [6]

The dependent task graph taken from Niyom et al. [6] is shown in Fig. 13 and the corresponding execution time $t_{p_a}(v_i)$ for each p_i and v_j are shown in Table 7. The values of $d_{p_a}(v_i)$ and $t_{p_a}(v_i)$ are also given in Table 7, where $v = 23$, $p = 4$, and $l = 6$. In addition, some processing constraints for security and other purposes were imposed. Firstly, each

Table 3
Execution time $t_{p_a}(v_i)$ of the task performed on each processing unit (test case 3).

Task	Execution time		
v_i	$t_{p_a}(v_i)$	$t_{p_b}(v_i)$	$t_{p_c}(v_i)$
1	19	41	34
2	28	46	20
3	36	34	62
4	15	25	37
5	30	50	54
6	33	35	59
7	12	20	21
8	13	22	24
9	41	68	73

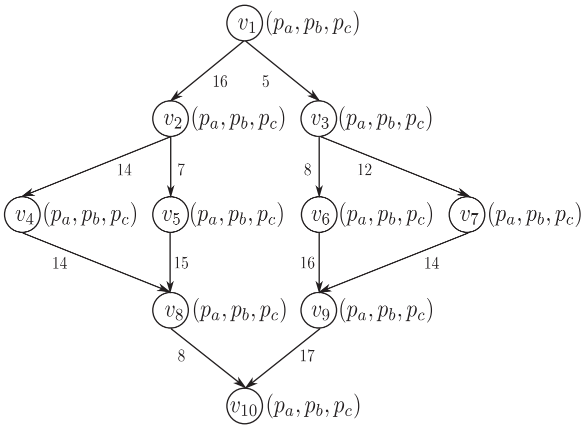

Fig. 11. A dependent task graph taken from Khan [26] with its transmission and execution costs (test case 4).

Table 4
Execution time $t_{p_a}(v_i)$ of the task performed on each processing unit (test case 4).

Task	Execution time		
v_i	$t_{p_a}(v_i)$	$t_{p_b}(v_i)$	$t_{p_c}(v_i)$
1	7	8	9
2	11	14	17
3	12	15	18
4	10	8	12
5	5	7	6
6	9	7	5
7	6	8	7
8	14	12	10
9	10	8	6
10	11	13	15

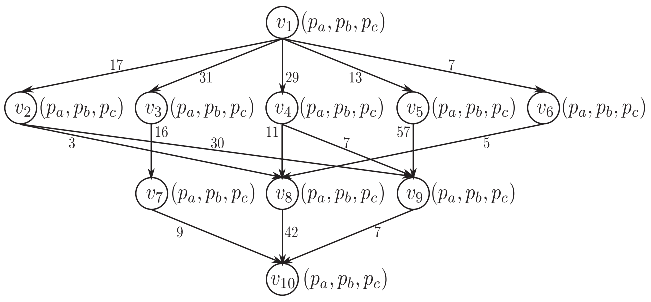

Fig. 12. A dependent task graph taken from Arabnejad and Barbosa [27] with its transmission and execution costs (test case 5).

Table 5
Execution time $t_{p_a}(v_i)$ of the task performed on each processing unit (test case 5).

Task	Execution time		
v_i	$t_{p_a}(v_i)$	$t_{p_b}(v_i)$	$t_{p_c}(v_i)$
1	22	21	36
2	22	18	18
3	32	27	43
4	7	10	4
5	29	27	35
6	26	17	24
7	14	25	30
8	29	23	36
9	15	21	8
10	13	16	33

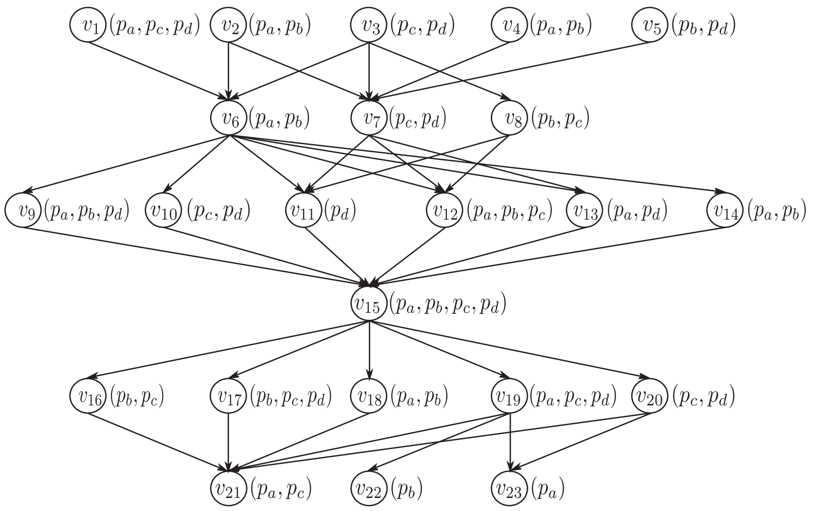

Fig. 13. A dependent task graph taken from Niyom et al. [6] with its transmission and execution costs (test case 6).

Table 6
Data transmission rate r_{p_a,p_b} for all processing unit pairs in unit amount of data per unit time (test case 6).

Processing Unit	p_a	p_b	p_c	p_d
p_a	–	1	0.5	1
p_b	1	–	0.5	1.25
p_c	0.5	0.5	–	1.25
p_d	1	1.25	1.25	–

processing unit can execute only some tasks due to security requirements and processing capability. Secondly, the data transmission delay $w_{p_a}(v_i)$ of each task depends on the transmission rate among connecting processing units and the amount of data $d_{p_a}(v_i)$ being transmitted. Table 6 shows the transmission rate of all processing unit pairs in unit amount of data per unit time. For example, processing unit p_a of Fig. 13 executes task v_1 and wants to send the data to processing unit p_b. The amount of data transmitted from p_a to p_b (from Table 7) is $d_{p_a}(v_1) = 25$ and data transmission rate from p_a to p_b (from Table 6) is $r_{p_a,p_b} = 1$. Thus, data transmission delay $w_{p_a}(v_1)$ is equal to $d_{p_a}(v_1)/r_{p_a,p_b} = 25/1 = 25$. The results of schedule length for HEFT, PETS, Lookahead, CEFT, PEFT, and PATS algorithms are 5230, 5220, 5230, 4962, 5526, and 4420, respectively.

Table 8 summarizes the schedule length of all compared scheduling algorithms from the above test cases. Both Lookahead and CEFT algorithms gave the shortest schedule length in one test case. The PETS algorithm provided the shortest schedule length in two test cases. PEFT and PATS algorithms yielded the shortest schedule length in four test cases. It is worth noting that the PATS algorithm yielded the schedule length close to the shortest schedule length in test cases 2 and 5. It is also near the optimal schedule in all test cases.

Table 7
Transmission data $d_{p_a}(v_i)$ of each task and execution time $t_{p_a}(v_i)$ required by each processing unit (test case 6).

Task	Amount of transmission data	Execution time			
v_i	$d_{p_a}(v_i)$	$t_{p_a}(v_i)$	$t_{p_b}(v_i)$	$t_{p_c}(v_i)$	$t_{p_d}(v_i)$
1	25	350	–	570	430
2	30	430	270	–	–
3	20	–	–	510	450
4	45	590	710	–	–
5	60	–	920	–	790
6	70	360	480	–	–
7	30	–	–	840	720
8	55	–	710	380	–
9	20	390	670	–	380
10	25	–	–	550	580
11	15	–	–	–	620
12	65	870	790	720	–
13	20	560	–	–	540
14	10	430	410	–	–
15	85	280	260	270	250
16	35	–	520	480	–
17	20	–	380	390	420
18	15	610	690	–	–
19	30	250	–	270	480
20	10	–	–	620	710
21	70	760	–	890	–
22	30	–	540	–	–
23	35	580	–	–	–

Table 8
Schedule length of all algorithms for all test cases.

Graph	HEFT	PETS	Lookahead	CEFT	PEFT	PATS
Case 1	80	76	76	81	85	**73**
Case 2	29	**27**	31	32	**27**	28
Case 3	260	257	**184**	224	**184**	**184**
Case 4	85	**73**	76	**73**	**73**	**73**
Case 5	133	147	127	136	**122**	124
Case 6	5230	5220	5230	4962	5526	**4420**

5.2. *Task graphs from real world applications*

In this experiment, all test cases were taken from the task graphs in real world problems which are Montage Work Flow, Molecular Dynamic Code, Gaussian Elimination, and Fast Fourier Transformation. The following evaluation parameters pertaining to these dependent task graphs are considered.

- *Number of task (v)*. This parameter is used to expand the size of a task graph by increasing the input matrix as employed in Gaussian Elimination and Fast Fourier Transformation. It is also used to evaluate the efficiency and the robustness of an algorithm by increasing the number of tasks.
- *Number of processing unit (p)*. This parameters is used to evaluate the efficiency of resource usage.
- *Communication to computation ratio (CCR)*. This parameter is the ratio of communication cost between tasks in the dependent task graph and the average of computation cost. If the *CCR* value is high, then it implies that the network speed is low or data congestion may occur in the network. On the other hand, if this value is low, then the network speed is high or there are small amount of data being transferred across the network.

Performance of PATS and all compared algorithms was measured by the comparison metrics given in Section 3.2. The result of each test case is the following.

Montage work flow

Montage is an astronomical image mosaic engine used to process larger-scale images such as science-grade mosaics of the sky. We took 50 tasks of Montage Work Flow to test the compared scheduling algorithm. In this case, the number of tasks was fixed at 50, the number of processing units ranged from 2 to 16, and the CCR value was from 0.1 to 10.0. The results are shown in Figs. 14 and 15. Fig. 14 shows the efficiency of compared algorithms when the number of processing units increased. From the figure, the PATS algorithm yielded the higher efficiency than the other compared algorithms for a low number of resources. When the resources of processing unit were increased, the efficiency of all algorithms drew

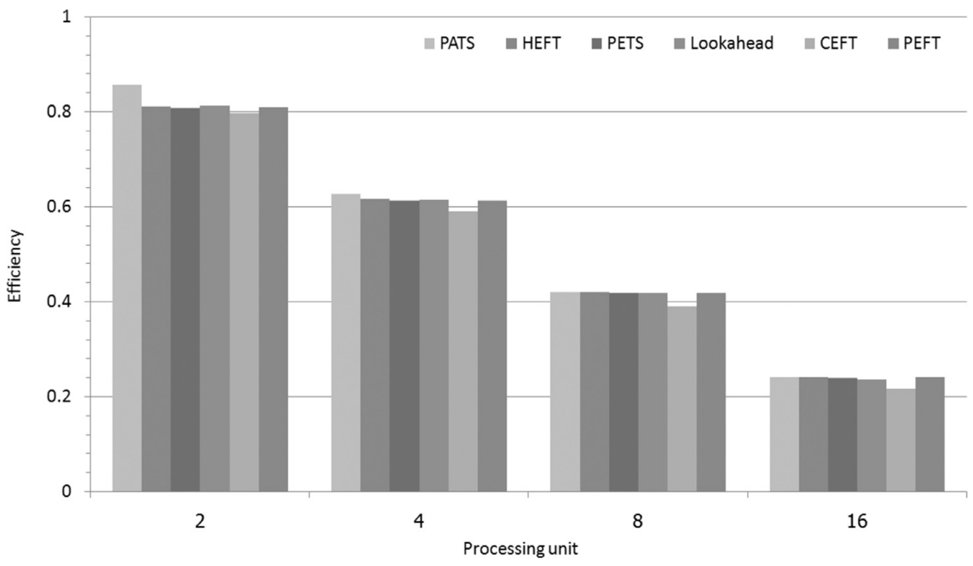

Fig. 14. Efficiency if compared algorithms for Montage Work Flow.

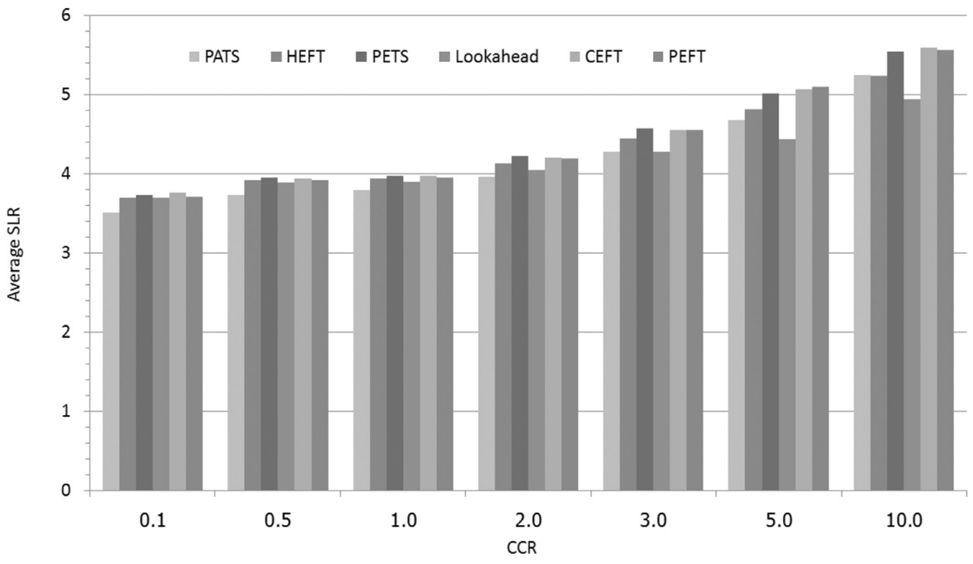

Fig. 15. Average SLR of compared algorithms for Montage Work Flow.

closer to that of PATS algorithm although the PATS algorithm showed a slightly higher efficiency. Fig. 15 shows the result of average SLR of all algorithms. From the figure, the PATS algorithm yielded the lowest average SLR for CCR value less than or equal to 3.0. However, when the value of CCR was greater than 3.0, the Lookahead algorithm yielded the lowest value, followed by PATS algorithm.

Molecular dynamic code

This task graph is a synthetic computation graph which was modified for molecular dynamic code. The graph was taken from the works of Kim and Browne [14] and Topcuoglu et al. [23]. The number of tasks was fixed at 40. The CCR value was from 0.1 to 10.0. The number of processing units was from 2 to 7 and the maximum number of tasks was seven in any level. The results are shown in Figs. 16 and 17. Fig. 16 shows the efficiency of compared algorithms when the number of resources or processing units were increased. From the figure, the PATS algorithm has the highest efficiency compared with the other algorithms. When the number of processing units was increased, the efficiency of all algorithms was similar although the PATS algorithm gave a slightly higher efficiency. Fig. 15 shows the result of average SLR of all algorithms. From the figure, the PATS algorithm gave the lowest average SLR for CCR value which was less than or equal to 5.0. However, when the value of CCR increased to 10.0, the Lookahead algorithm yielded the lowest average SLR, followed by PETS and PATS algorithms.

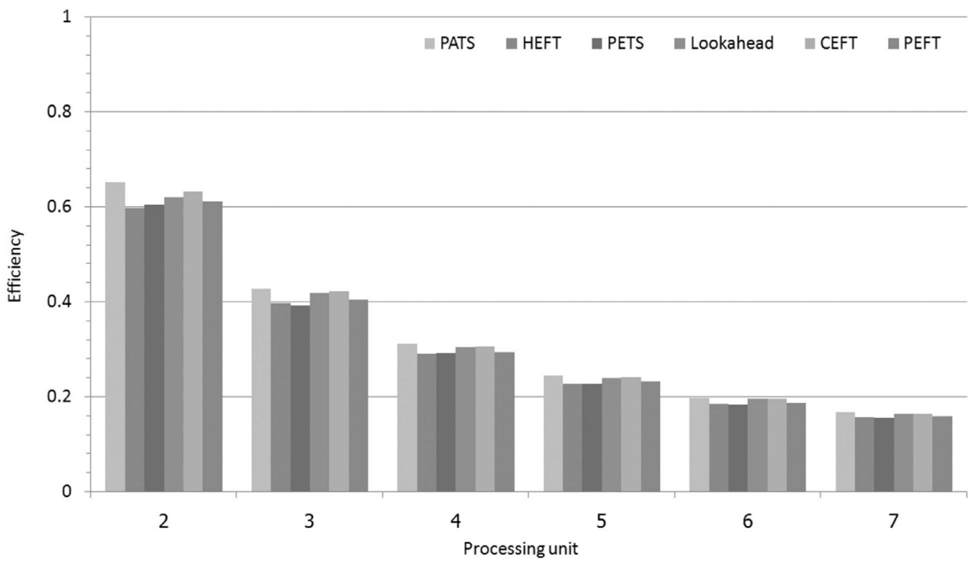

Fig. 16. Efficiency of compared algorithms for Molecular Dynamic Code.

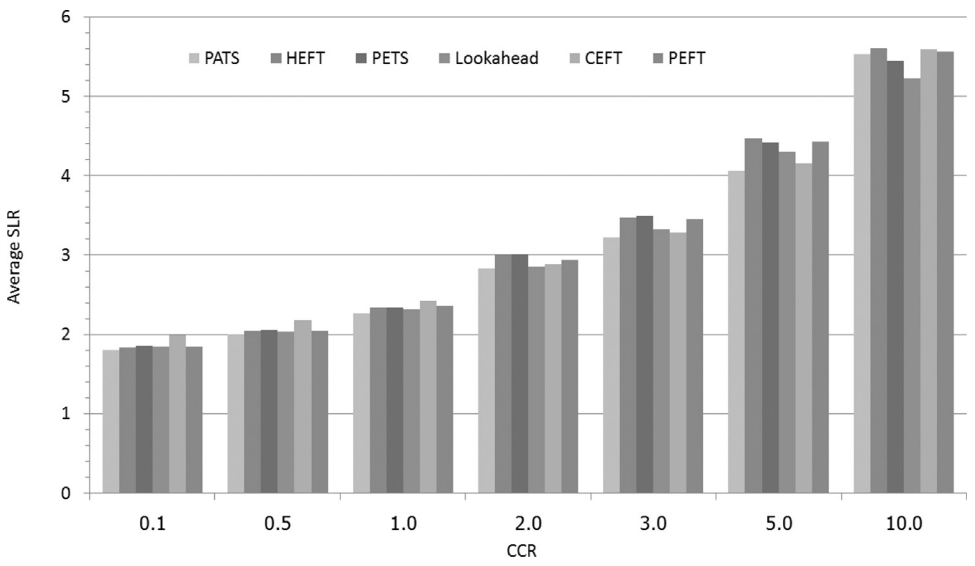

Fig. 17. Average SLR of compared algorithms for Molecular Dynamic Code.

Fast Fourier Transformation

Fast Fourier transformation graph has a total number of tasks equals to $(2 \times m - 1) + (m \times \log_2 m)$, where the input matrix $m = 2^k$ and k is an integer. The first term is the number of recursive call tasks and the second term is the number of butterfly operation tasks. For this experiment, the value of m ranged from 4 to 256. Input tasks were from 15 to 2559. The number of the processing units was from 2 to 16, and CCR value was from 0.1 to 10.0. The results are shown in Figs. 18–23. Fig. 18 shows the result average SLR of all algorithms where the CCR value increased from 0.1 to 10.0, and the input matrix was fixed at 32. From the figure, the PATS algorithm yielded the lowest average SLR for all CCR values.

Fig. 19 shows the result average SLR when the number of processing units was fixed at 3, and the number of input matrices increased from 4 to 256. From the figure, the PATS algorithm yielded the lowest average SLR for all number of input matrix sizes. The algorithm running time of this case was shown in Fig. 20. From the figure, where the number of input matrices was less than 16, all compared algorithms running time took less than 1 second. The PATS algorithm spent less running time than other compared algorithms when the number of input matrices was 32. When increasing the number of input matrices to 256, the PATS algorithm spent at least 82 times less running time than other compared algorithms.

Fig. 21 shows the result efficiency when the number of input matrix was fixed at 128, 1151 tasks, and the number of processing units increased from 2 to 16. From the figure, when the number of processing units was 2, the efficiency of PATS

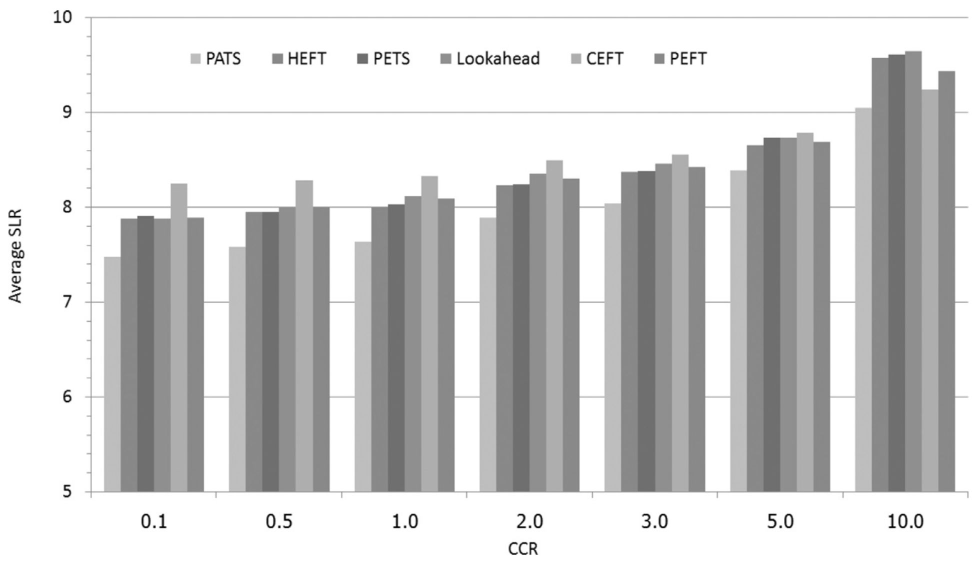

Fig. 18. Results of average SLR as a function of CCR from compared algorithms for Fast Fourier Transformation.

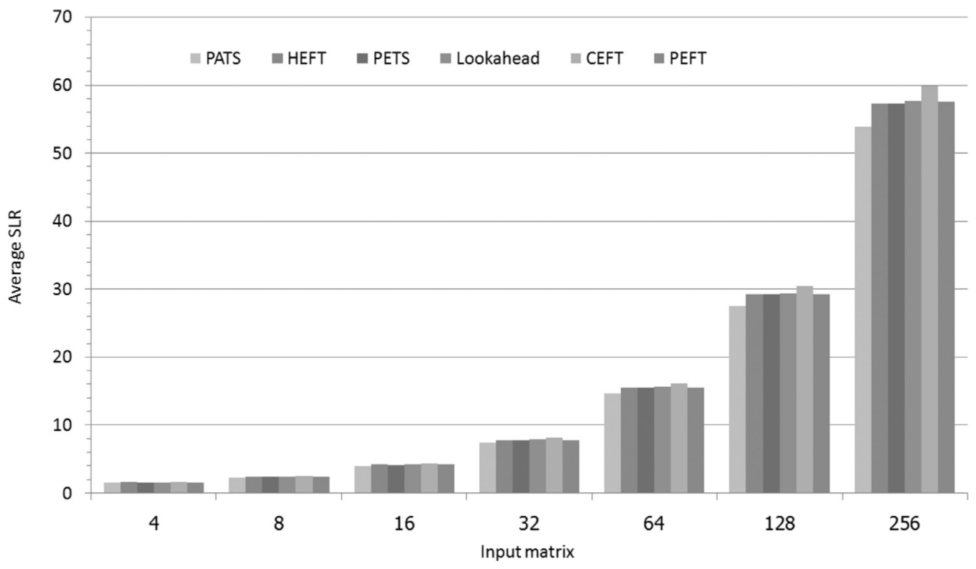

Fig. 19. Results of average SLR as a function of number of input matrices from compared algorithms for Fast Fourier Transformation.

algorithm was higher than other compared algorithms. When the number of processing units was 16, the efficiency of most algorithms drew closer which was less than 1% apart. The algorithm running time and schedule length were determined under the same condition. In Fig. 22, the correlation between the results of the two metrics are presented. From the figure, when the number of processing unit was 16, the schedule lengths of most algorithms were longer than that of PATS algorithm but the values were close. The yielded running time of the PATS algorithm, however, was an order of magnitude less than those of compared algorithms.

Given the execution cost of each task is measured in seconds, the performance percentage (with reference to HEFT) of all compared algorithms are shown in Fig. 23. From the figure, the performance percentage of PATS algorithm is greater than 100. This is because both the scheduling length and running time of PATS algorithm were lower than those of the HEFT algorithm. When the number of processing unit was increased to 16, the performance percentage of PATS algorithm was substantially higher than other compared algorithms which is due to the much lower running time. Furthermore, the PATS algorithm yielded the lowest system finish time for 2 to 16 processing units as shown in Table 9.

Gaussian Elimination

Gaussian elimination graph contains a total number of task equals to $(m^2 + m - 2)/2$, where m denotes the number of input matrices. For this experiment, the number of input matrices ranged from 10 to 70. Input tasks were from 54 to 2484.

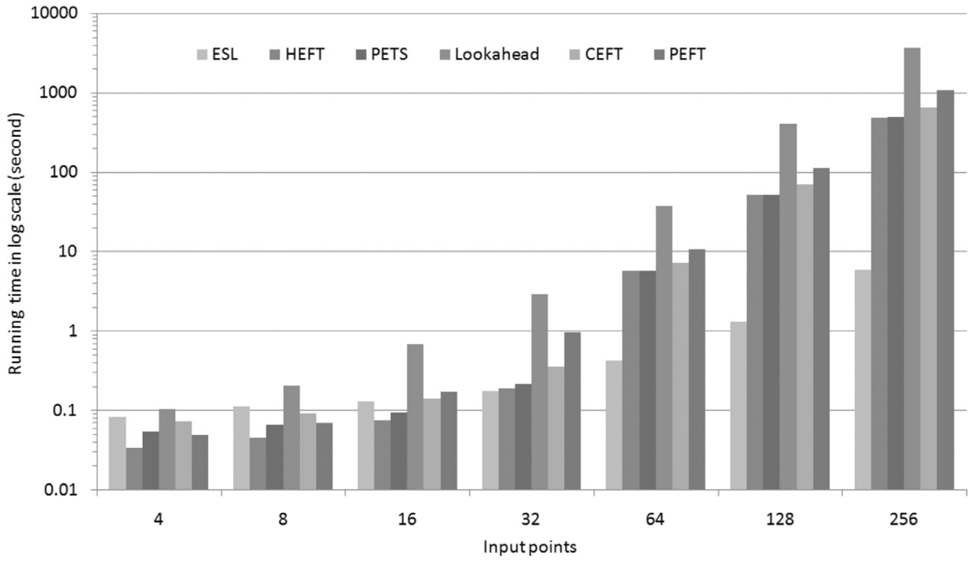

Fig. 20. Results of algorithm running time as a function of the number of input matrices from compared algorithms for Fast Fourier Transformation.

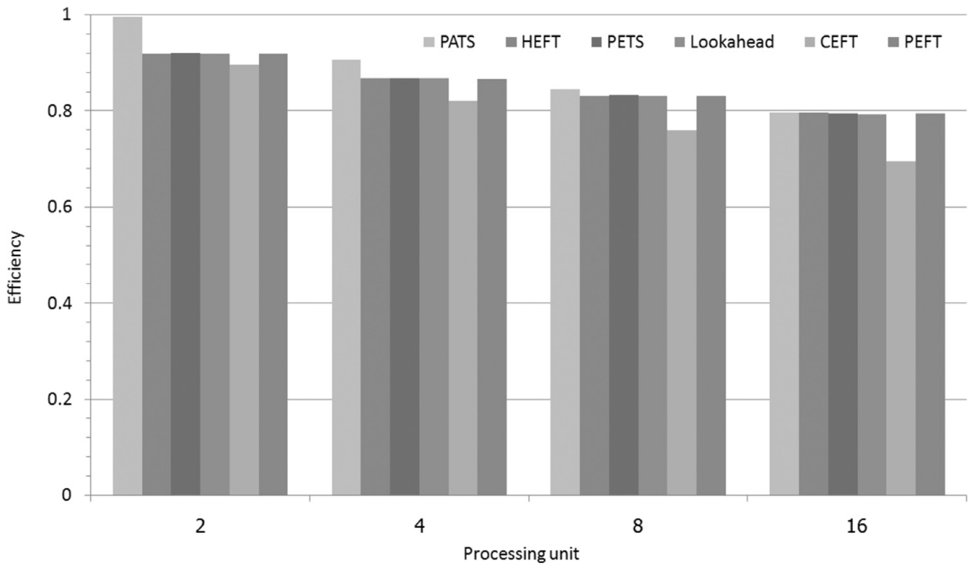

Fig. 21. Results of efficiency as a function of the number of processing units from compared algorithms for Fast Fourier Transformation.

Processing units were from 2 to 16 and *CCR* was from 0.1 to 10.0. The results are shown in Figs. 24–29. Fig. 24 shows the results of average SLR of all algorithms, where the CCR value increased from 0.1 to 10.0 and the number of input matrix was fixed at 30. From the figure, PATS algorithm yielded the lowest average SLR for all CCR values which was similar to the Fast Fourier Transformation case.

Fig. 25 shows average SLR where the number of processing units was fixed at 3 and the number of input matrices increased from 10 to 70. From the figure, the PATS algorithm yielded the lowest average SLR where the number of input matrices was from 10 to 70. The obtained average SLR was at least 5% lower than all compared algorithms as the matrix input went from 70 or 2484 tasks. From running time comparison in Fig. 26, the PATS algorithm spent less running time less than other compared algorithms as the input matrix was greater than or equal to 30 or 464 tasks. Finally, as the input became 2484 tasks, the running time of PATS algorithm was at least 36 times less than other compared algorithms.

Fig. 27 shows the efficiency where the number of input matrices was fixed at 50 or 1274 tasks and the number of processing units increased from 2 to 16. From the figure, the PATS algorithm yielded the highest efficiency for 2 to 16 processing units. Fig. 28 shows the combined result from the algorithm running time and schedule length metrics. From the figure, the PATS algorithm yielded the lowest running time and schedule length. Furthermore, the PATS algorithm yielded

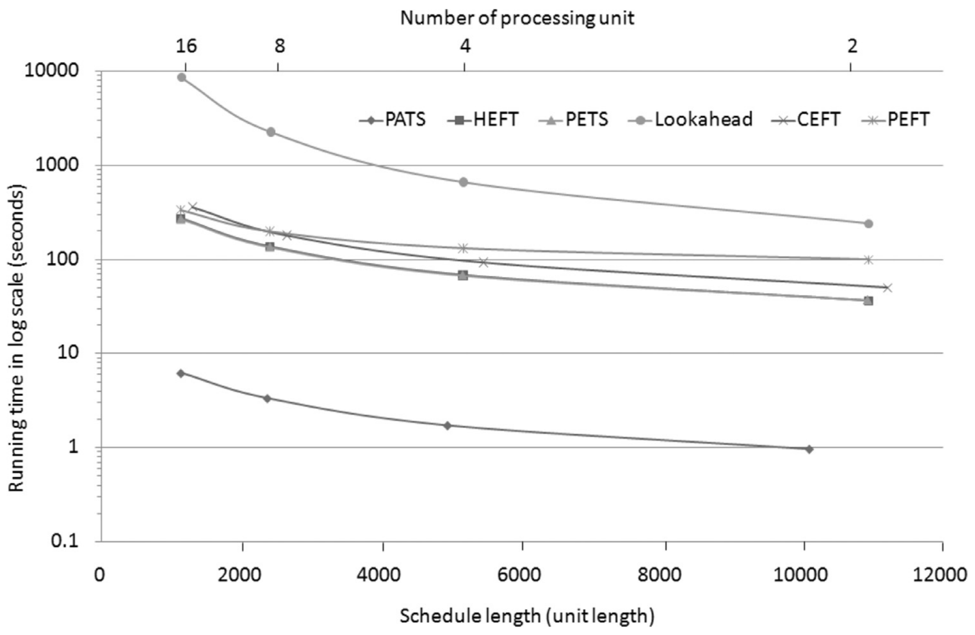

Fig. 22. Results of correlation between algorithm running time and schedule length on a different number of processing units from compared algorithms for Fast Fourier Transformation.

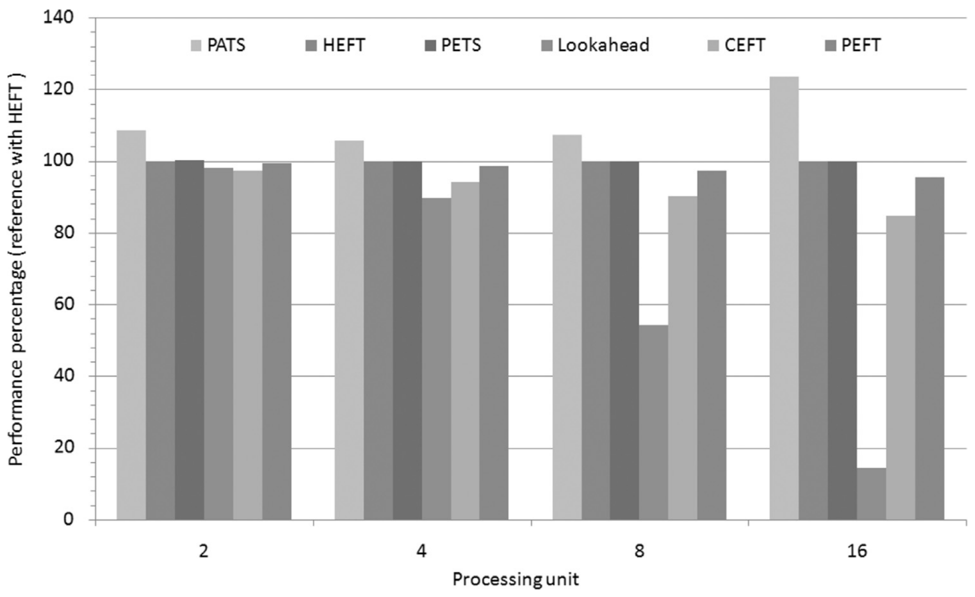

Fig. 23. Results of performance percentage (reference with HEFT) as a function of the number of processing units from compared algorithms for Fast Fourier Transformation.

Table 9
System finish time of all algorithms for Fast Fourier Transformation.

Processing unit	PATS	HEFT	PETS	Lookahead	CEFT	PEFT
2	**10087**	10976.33	10953.50	11176.77	11259.26	11039.84
4	**4932**	5221.53	5215.38	5812.66	5537.33	5291.51
8	**2366**	2540.32	2537.66	4676.20	2815.52	2606.28
16	**1136**	1404.49	1402.88	9707.96	1658.27	1468.07

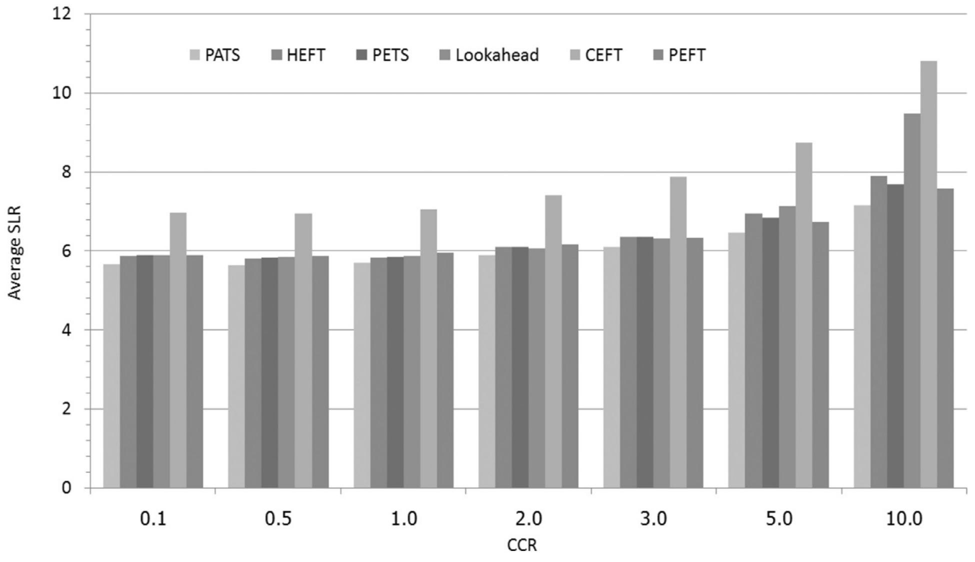

Fig. 24. Results of average SLR as a function of CCR from compared algorithms for Gaussian Elimination.

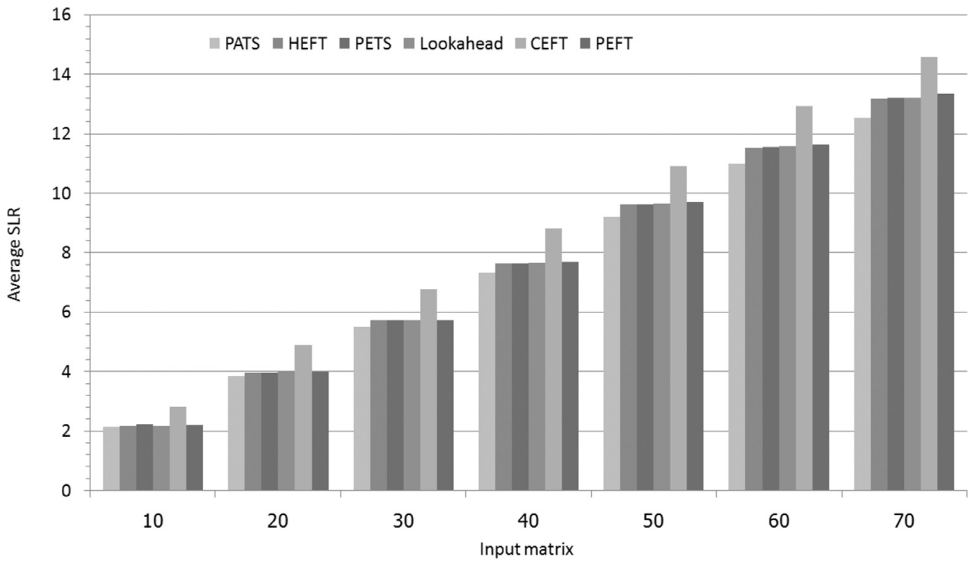

Fig. 25. Results of average SLR as a function of number of input matrices from compared algorithms for Gaussian Elimination.

Table 10
System finish time of all algorithms for Gaussian Elimination.

Processing unit	PATS	HEFT	PETS	Lookahead	CEFT	PEFT
2	**11317**	12191.31	12173.58	12449.51	13034.30	12304.47
4	**5564**	5834.93	5842.77	6720.68	6940.61	5968.95
8	**2745**	2975.31	2981.00	6307.50	4052.75	3089.53
16	**1513**	1954.54	1955.02	14488.25	2877.47	2002.71

the highest performance percentage (with reference to HEFT) for 2 to 16 processing units as shown in Fig. 29. Moreover, the PATS algorithm yielded the lowest system finish time for 2 to 16 processing units as shown in Table 10.

6. Discussion

The proposed PATS algorithm accomplishes two essential performance attributes of task scheduling on heterogeneous systems, namely, level-based task assignment by using EFT candidate task list and idle slot reduction. From the experimental

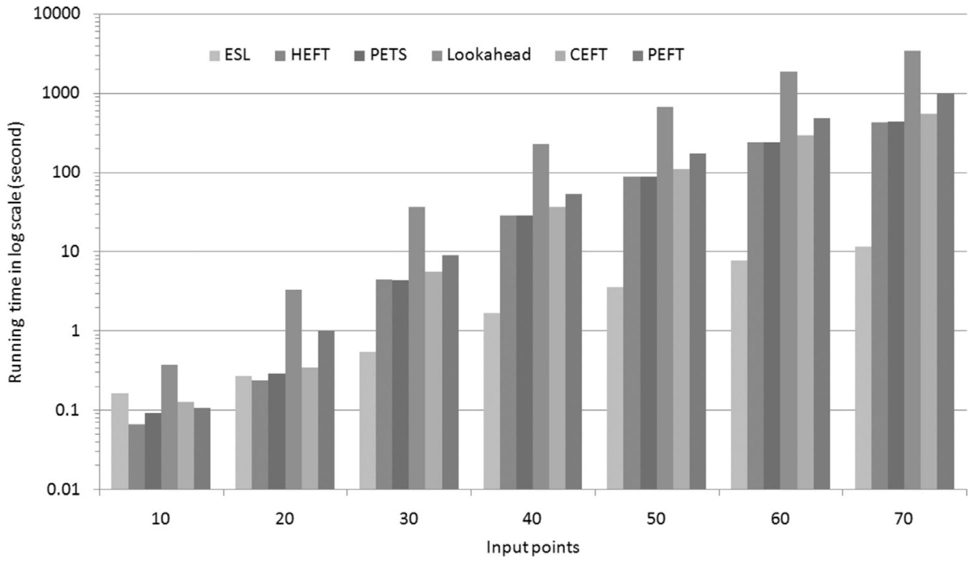

Fig. 26. Results of algorithm running time as a function of the number of input matrices from compared algorithms for Gaussian Elimination.

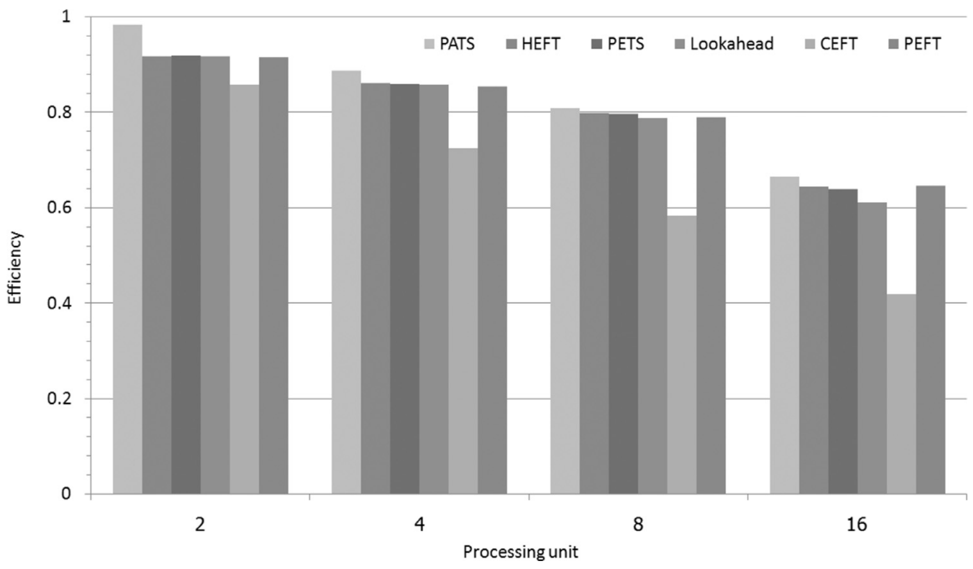

Fig. 27. Results of efficiency as a function of the number of processing units from compared algorithms for Gaussian Elimination.

results, the PATS algorithm gave the fastest running time in all cases. The advantage became more apparent in the case where the number of computational elements increased as shown in Figs. 20, 22, 23, 26, 28, and 29. Such results were attributed by the algorithmic low time complexity, which will be explained below.

The time complexity of PATS algorithm is determined by finding an EFT candidate task list on each level of the dependent task graph. Suppose the tasks are spread evenly on each level of the graph. The average number of tasks on each level must not exceed v/l. The amount of time required to compute an EFT of one task depends on the number of its predecessor tasks in the preceding level and the number of candidate processing units itself. This means that the number of elements considered is $(v/l) \times p$. Thus, the number of elements to be considered to compute the EFT candidate task list for all tasks on a given level is equal to $(v/l)^2 \times p$. Therefore, the overall time complexity to compute the EFT candidate task list for all levels step is equal to $(v^2 \times p)/l$.

In the idle slot reduction step, the number of tasks to be considered and inserted to an idle slot cannot be greater than l. The complexity for rescheduling of inserted one task is equal to v^2/l. Thus, the overall time complexity for the idle slot reduction step is equal to $O(v^2)$. The combined time complexity in all steps becomes $max(O((v^2 \times p)/l), O(v^2))$, where v, l, and p are defined earlier in Section 3. Notice that the above time complexities are seemingly not far apart from those of the compared algorithms given in Section 2. Nonetheless, division of the term l curbs the growth effects of other parameters.

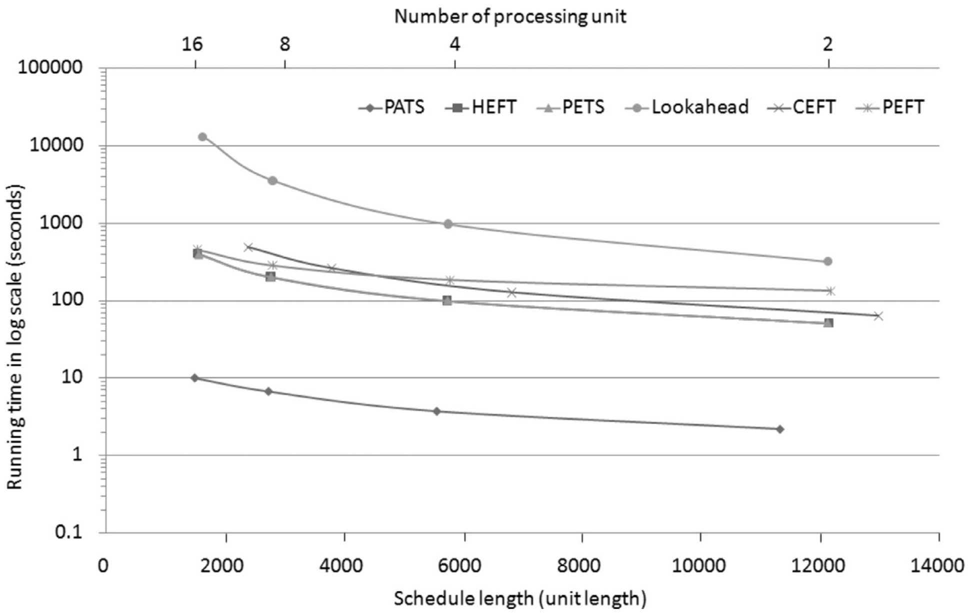

Fig. 28. Results of correlation between algorithm running time and schedule length on a different number of processing units from compared algorithms for Gaussian Elimination.

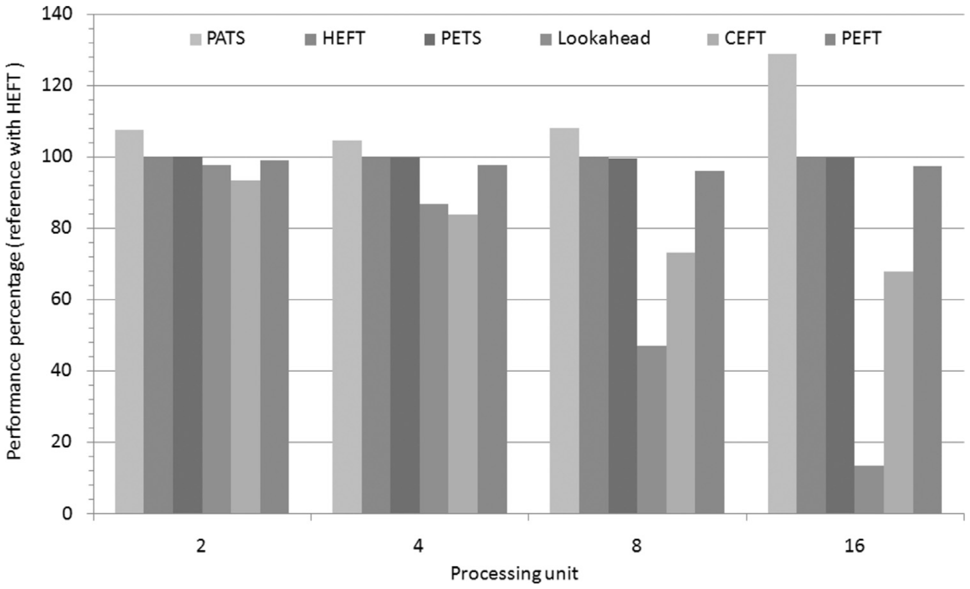

Fig. 29. Results of performance percentage (reference with HEFT) as a function of the number of processing units from compared algorithms for Gaussian Elimination.

Consequently, the system finish time of the real world applications produced by PATS are lower than the time produced by all compared algorithms.

7. Conclusion

The proposed PATS distributed assignment algorithm focuses on low time complexity having shortest schedule length or makespan and idle reduction. The key concept of PATS algorithm relies on how it determines the candidate tasks one level at a time to arrive at predicting and selecting proper task to create an EFT candidate task list. Hence, the schedule will match the resource of available processing units. In the meantime, idle slot reduction is used to reduce waiting time on a processing unit. Thereby the schedule length has less time complexity. The study employed dependent task graphs from all compared algorithms as well as real world application task graphs to see how PATS significantly achieved. The performance evaluation was based on the measure of schedule length, average schedule length ratio, running time of algorithm, efficiency, and performance percentage metrics. The experimental results showed that the PATS algorithm yielded the best schedule length in most cases of scheduling experiment. For real world applications, the PATS algorithm yielded the best efficiency, system finish time, and performance percentage in all cases and the best average schedule ratio in most cases. Furthermore, the running time performance of PATS algorithm was obviously high when the number of task increased. The significant benefits of PATS algorithm over the compared algorithms are lower time complexity and shorter schedule length. One of the important issues to be considered in future work is the energy expended by each processing unit and the total energy saving in a real distributed heterogeneous environment.

Acknowledgments

This work is supported by the Office of the Higher Education Commission, Ministry of Education, Thailand.

References

[1] J. Khan, R. Vemuri, An iterative algorithm for battery-aware task scheduling on portable computing platforms, Proceedings of the Design, Automation and Test in Europe (DATE), IEEE Computer Society, Munich, Germany, 2005, pp. 622–627.
[2] V. Rao, N. Navet, G. Singhal, A. Kumar, G.S. Visweswaran, Battery aware dynamic scheduling for periodic task graphs, Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Rhodes Island, Greece, 2006.
[3] S. Zhang, K.S. Chatha, Automated techniques for energy efficient scheduling on homogeneous and heterogeneous chip multi-processor architectures, Proceedings of the 13th Asia and South Pacific Design Automation Conference (ASPDAC), IEEE, Seoul, Korea, 2008, pp. 61–66.
[4] C. Yang, J. Chen, T. Kuo, L. Thiele, An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), IEEE, Nice, France, 2009, pp. 694–699.
[5] S. Zikos, H.D. Karatza, Performance and energy aware cluster-level scheduling of compute-intensive jobs with unknown service times, Simul. Model. Pract. Theory 19 (1) (2011) 239–250.
[6] A. Niyom, P. Sophatsathit, C. Lursinsap, An energy-efficient process clustering assignment algorithm for distributed system, Simul. Model. Pract. Theory 40 (2014) 95–111.
[7] J. Li, M. Qiu, J. Niu, T. Chen, Battery-aware task scheduling in distributed mobile systems with lifetime constraint, Proceedings of the 16th Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, Yokohama, Japan, 2011, pp. 743–748.
[8] X. Jin, F. Zhang, Y. Song, L. Fan, Z. Liu, Energy-efficient scheduling with time and processors eligibility restrictions, Proceedings of the 19th International European Conference on Parallel and Distributed Computing (Euro- Par), Springer Berlin Heidelberg, Aachen, Germany, 2013, pp. 66–77.
[9] G. Terzopoulos, H.D. Karatza, Performance evaluation and energy consumption of a real-time heterogeneous grid system using DVS and DPM, Simul. Model. Pract. Theory 36 (2013) 33–43.
[10] L. Benini, A. Bogliolo, G.D. Micheli, A survey of design techniques for system-level dynamic power management, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 8 (3) (2000) 299–316.
[11] G.L. Stavrinides, H.D. Karatza, Scheduling multiple task graphs in heterogeneous distributed real-time systems by exploiting schedule holes with bin packing techniques, Simul. Model. Pract. Theory 19 (2011) 540–552.
[12] G.L. Stavrinides, H.D. Karatza, Scheduling real-time DAGs in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes, Future Gener. Comput. Syst. 28 (7) (2012) 977–988.
[13] A.K. Amoura, E. Bampis, J.C. Konig, Scheduling algorithms for parallel Gaussian elimination with communication costs, IEEE Trans. Parallel Distrib. Syst. 9 (7) (1998) 679–686.
[14] S.J. Kim, J.C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, in: Proceedings of the International Conference on Parallel Processing, 1988, pp. 1–8.
[15] S. Darbha, D.P. Agrawal, Optimal scheduling algorithm for distributed-memory machines, IEEE Trans. Parallel Distrib. Syst. 9 (1) (1998) 87–95.
[16] V. Liberatore, Multicast scheduling for list requests, Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), IEEE, San Francisco, California, USA, 2002, pp. 1129–1137.
[17] I. Ahmad, Y. Kwok, On exploiting task duplication in parallel program scheduling, IEEE Trans. Parallel Distrib. Syst. 9 (9) (1998) 872–892.
[18] D. Bozdag, U. Catalyurek, F. Ozguner, A task duplication based bottom-up scheduling algorithm for heterogeneous environments, Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS), IEEE, Rhodes Island, Greece, 2006.
[19] T. Yang, A. Gerasoulis, DSC: scheduling parallel tasks on an unbounded number of processors, IEEE Trans. Parallel Distrib. Syst. 5 (9) (1994) 951–967.
[20] J. Liou, M.A. Palis, A comparison of general approaches to multiprocessor scheduling, Proceedings of the 11th International Parallel Processing Symposium (IPPS), IEEE Computer Society, Geneva, Switzerland, 1997, pp. 152–156.
[21] R.C. Correa, A. Ferreira, P. Rebreyend, Integrating list heuristics in to genetic algorithm for multiprocessing scheduling, Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, IEEE, New Orleans, USA, 1996, pp. 462–469.
[22] T.D. Braun, H.J. Siegal, N. Beck, L.L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen, R.F. Freund, A comparison study of static mapping heuristics for a class of meta-task in heterogeneous computing system, in: Proceedings of the 8th Heterogeneous Computing Workshop, San Juan, Puerto Rico, 1999, pp. 15–29.
[23] H. Topcuoglu, S. Hariri, M. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[24] E. Ilavarasan, P. Thambidurai, Low complexity performance effective task scheduling algorithm for heterogeneous computing environments, J. Comput. Sci. 3 (2) (2007) 94–103.
[25] L.F. Bittencourt, R. Sakellariou, E.R.M. Madeira, DAG scheduling using a Lookahead variant of the heterogeneous earliest finish time algorithm, Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP), IEEE Computer Society, Pisa, Italy, 2010.
[26] M.A. Khan, Scheduling for heterogeneous systems using constrained critical paths, Parallel Comput. 38 (2012) 175–193.
[27] H. Arabnejad, J.G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2014) 682–694.