

Task scheduling for heterogeneous computing systems

Shaikhah AlEbrahim¹ · Imtiaz Ahmad¹

Published online: 9 November 2016

© Springer Science+Business Media New York 2016

Abstract Efficient scheduling of tasks in heterogeneous computing systems is of primary importance for high-performance execution of programs. The programs are to be considered as multiple sequences of tasks that are presented as directed acyclic graphs (DAG). Each task has its own execution timeline that incorporates into multiple processors. Moreover, each edge on the graph represents constraints between the sequenced tasks. In this paper, we propose a new list-scheduling algorithm that schedules the tasks represented in the DAG to the processor that best minimizes the total execution time by taking into consideration the restriction of crossover between processors. This objective will be achieved in two major phases: (a) computing priorities of each task that will be executed, and (b) selecting the processor that will handle each task. The first phase, priorities computation, focuses on finding the best execution sequence that minimizes the makespan of the overall execution. In list-scheduling algorithm, the quality of the solution is very sensitive to the priority assigned to the tasks. Therefore, in this paper, we include an enhanced calculation of weight that is used in the ranking equation for determining the priority of tasks. The second phase, processor selection, primarily focuses on allocating a processor that is a best fit for each task to be executed. In this paper, we enhance the processor selection by introducing a randomized decision mechanism based on a threshold which decides whether the task be assigned to the processor with the lowest execution time or to the processor that produces the lowest finish time. This mechanism considers a balanced combination of the local and global optimal results to explore the search space efficiently to optimize the over-

✉ Imtiaz Ahmad
imtiaz.ahmad@ku.edu.kw

Shaikhah AlEbrahim
shaikha_alebrahim@hotmail.com

¹ Computer Engineering Department, Kuwait University, P.O. Box 5969, Safat 1306, Kuwait

all makespan. The proposed algorithm is evaluated on different randomly generated DAGs, and results are compared with the well-known existing approaches to show the effectiveness of the proposed algorithm in reducing the makespan of execution. The experiment's results show improvement in the makespan that reaches up to 6–7%.

Keywords Directed acyclic graphs · Heterogeneous systems · List scheduling · Task ranking · Task scheduling

1 Introduction

Recent years have witnessed an exponential growth in the popularity and development of cloud computing systems and services. As an emerging and promising computing paradigm, cloud computing services are becoming the primary source of computing power for business, personal, and mobile computing applications. According to a report by the International Data Corporation [1], the worldwide market in public cloud computing services will be worth \$100 billion by 2016. Cloud computing providers rely on a shared pool of servers and consolidated data centers to offer their customer's subscription-based, on-demand, and easily accessible computation, communication, and storage resources in a scalable manner. Cloud computing environments are heterogeneous computing in nature where typically multi-core processors equipped with dedicated accelerators such as GPUs are interconnected with high-speed links to perform different compute intensive applications that have diverse computational requirements. As competition between cloud service providers intensifies and prices decrease, providers will need to reduce their expenses and increase service revenues, predominantly by enhancing the utilization of hardware resources. Task scheduling which is a process of mapping tasks to resources plays a critical role in resources utilization. Therefore, this has stirred a renewed interest in task scheduling for heterogeneous computing systems [2].

Task-scheduling approaches for homogeneous multiprocessor systems including the ones for reconfigurable network topologies have been reported to solve computationally intensive image processing and computer vision applications [3–10]. However, heterogeneity in computing systems introduces an additional degree of complexity to the scheduling problem as compared with homogeneous multiprocessor systems [2]. During assignment of a task to a processor, the scheduling technique has also to take into account the varying speed of processors. For the effective utilization of heterogeneous computing systems, an application program can be partitioned into a set of tasks (program segments) represented by an edge-weighted directed acyclic graph (DAG), such that each task is computationally homogeneous and can be assigned to the best suited processor. The performance of a parallel application on heterogeneous processors is highly dependent on both the application characteristics (execution cost, data dependencies between tasks, etc.), and platform features (computation capacity of the processors, number of processors, communication bandwidth, memory size, etc.). To effectively exploit heterogeneous computing systems, it is desired to achieve a minimum schedule length (makespan). The idea behind the minimization of the schedule

length is to maximize the system throughput and increase processor utilization to reduce cost [2, 11–13].

Extant literature has proven that DAG scheduling problem is NP-complete to obtain the minimum makespan [2, 14–20]. The widely known task-scheduling problem is mainly classified into two categories: static scheduling and dynamic scheduling [21–23]. In static scheduling, all information about tasks, such as execution and communication times, is known beforehand. On the other hand, in dynamic scheduling, execution and communication times are known at runtime. After executing each task, the new values of the next level are calculated. Static scheduling algorithms are classified into two major groups: heuristic-based and guided random search-based algorithms. The former algorithm provides near-optimal results with polynomial time complexity and acceptable performance. Alternatively, the latter gives optimal solutions with exponential time complexity. The heuristic-based algorithms are further categorized into list, clustering, and duplication scheduling. The duplication heuristics produce the shortest makespan, but its disadvantages are that they have high time complexity and lower efficiency [23]. On the other hand, list-scheduling heuristics produce the most efficient schedules without compromising the overall makespan [14, 18, 21, 24, 25].

Among many different scheduling approaches, list-scheduling heuristics have been proven to produce the most efficient schedules with a complexity that is, in general, quadratic in relation to the number of tasks [21, 26]. In addition, list-scheduling algorithms are very fast, easy to implement and have broader applicability. List-scheduling consists of two major steps: task prioritization and task allocation to selected processor. In the task prioritization step, each task is assigned a priority based on its estimated importance to determine the entire schedule. Initially, a ready list holds all the tasks which have no predecessors. A task with the highest priority is selected from the ready list and assigned to the available processor based on the processor selection mechanism. Different list-scheduling algorithms differ in these two steps and leads to different schedules. The priority assignment to tasks in list scheduling is critical in determining a good schedule. Accurate priority determination has led to a great deal of research to design efficient heuristics. Some of the most known list-scheduling algorithms are Heterogeneous Earliest Finish Time (HEFT) [19], Predict Earliest Finish Time (PEFT) [14], Heterogeneous Critical Parent Trees (HCPT) [27], High-Performance Task Scheduling (HPS) [28], and Performance Effective Task Scheduling (PETS) [29]. HEFT is known to generate a schedule length comparable to the schedule lengths of other scheduling algorithms with lower time complexity. The HEFT algorithm has been enhanced through introducing new mechanisms of calculating task prioritization and through adding new mechanisms (like the look-ahead attribute) to improve the processor's selection [30, 31].

In this paper, we introduce an enhancement to both the prioritization and the processor selection for the static list-based task-scheduling problem. Introducing this mechanism reduces the total makespan of the schedule compared to previously proposed algorithms. Moreover, our proposed algorithm has the same complexity in comparison to HEFT [19] and PEFT [14], which are the most well-known algorithms. The algorithm used in this study is $O(t^2p)$, where t represents the number of tasks and p represents the number of processors that execute these tasks. Having the

same overall complexity, the proposed algorithm produces more efficient schedules by minimizing the overall makespan of tasks being processed by a set of processors.

The remainder of the paper is organized as follows. Section 2 describes the problem formulation. In Sect. 3, we present extant work related to list scheduling for heterogeneous computing systems. Section 4 presents the proposed algorithm and includes an example that compares the results of previously reported state-of-the-art algorithms and the proposed algorithm. Section 5 discusses the experimental results of the algorithms. Finally, conclusions and possible future work are discussed in Sect. 6.

2 Problem formulation

In this section, we describe the non-preemptive static task-scheduling problem for heterogeneous computing systems, such as CPU–GPU platforms, with the objective of minimizing the overall execution time. An application program consists of a set of tasks and is modeled by a directed acyclic graph (DAG). In the graph, $G = (T, <, E)$, where $T = \{t_i, i = 1, \dots, n\}$ is a set of n tasks. $<$ represents a partial order on T . For any two tasks $(t_i \text{ and } t_j) \in T$, the existence of the partial order, $t_i < t_j$ means that task t_j cannot be scheduled until task t_i has been completed. Hence, t_i is a predecessor of t_j and t_j is a successor of t_i . E is the set of directed edges or arcs. A weight, c_{ij} , is associated with each arc that represents the amount of data to be transferred from task t_i to task t_j in bytes [32].

A heterogeneous computing system consists of a set of processors $P = \{P_j : j = 0, \dots, m-1\}$ interconnected by a fully connected topology. The Estimated Computation Time (*ECT*) of a task, t_i , on a processor, P_j , is denoted as $w_{i,j}$, where $0 \leq i < n$ and $0 \leq j < m$. The $w_{i,j}$ value of a task may be different on different processors depending on its computational capability. For static task scheduling, the $w_{i,j}$ value for each task–processor pair is assumed to be available a priori, which can be determined using code-type profiling. An example of a directed acyclic graph (DAG) consisting of 10 tasks and a heterogeneous system consisting of 2 processors is shown in Fig. 1. Furthermore, we assume that each processor in the heterogeneous system can perform computation and communication simultaneously. The communication cost is zero when two tasks, t_i, t_j are assigned to the same processor. Otherwise, data must be transferred from the processor on which task t_i is assigned to the processor where task t_j is assigned [21].

Some of the terminologies that will be referred to further in this paper are the *pred*(n_i), *succ*(n_i), *makespan*, *level*(n_i), *Critical Path*, *EST*(n_i, p_j) and *EFT*(n_i, p_j). *Pred*(n_i) represents the immediate predecessors of node n_i in the input DAG. The entry node is a node with no predecessors. *Succ*(n_i) represents the immediate successors of task n_i . The exit node has no successors. Note that t_i and n_i will be used interchangeably to represent a task. Furthermore, the makespan refers to the total schedule length. That means the makespan is equal to the maximum Actual Finish Time (AFT) of the exit node in the DAG. Therefore, makespan is defined by

$$\text{makespan} = \max \{AFT(n_{\text{exit}})\}$$

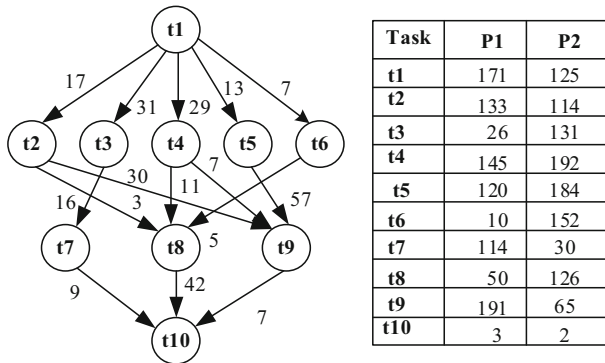


Fig. 1 Example of a DAG and the Estimated Computation Time, w_i, j , of the tasks on each processor

The maximum number of edges between a node and the entry node represents the level of a task in the DAG. For the entry node $level(n_{entry})$ is equal to 0. Furthermore, the longest path between the entry and the exit node is called the critical path of the DAG. The earliest start time of a node ni on processor pj is referred to as

$$EST(ni, pj) = \max \left\{ T_{available}(pj), \max_{nm \in pred(ni)} \{ AFT(nm) + cm, i \} \right\}$$

where $T_{available}(pj)$ is the time where a processor pj is ready to execute a new task. Furthermore, the inner maximum block is the total time that the processor needs to transfer all required data. For the entry node, since all processors are ready to execute and there are no predecessors, then $EST(n_{entry}, pj)$ is always equal to zero.

On the other hand, $EFT(ni, pj)$ represents the Earliest Finish Time of a task, ni , on processor pj . Moreover, EFT is the total of the EST of the processor and the computational cost of the task on the same processor. Therefore, $EFT(ni, pj)$ is defined by

$$EFT(ni, pj) = EST(ni, pj) + w_i, j$$

We assume that an input DAG has a single entry node and a single exit node. Therefore, if a DAG has more than one single entry or exit node, the two nodes are connected to an extra imaginary parent or child node respectively. In summary, the task-scheduling problem is to assign tasks to be executed by processors in the best way possible such that all above constraints are met in addition to having the minimal overall time to execution (makespan).

3 Related work

Task-scheduling algorithms for homogeneous systems have been well explored by previous researches [3–10]. In this section, we present a brief description of some of the previously proposed algorithms that focus on finding suboptimal solutions to

the problem of list-based scheduling heuristics. Some of the well-known previous heuristics that are going to be described are HEFT [19], PEFT [14], HCPT [27], HPS [28], PETS [29] and HCPPEFT [16].

The Heterogeneous Earliest Finish Time (HEFT) is one the best heuristics proposed by Topcuoglu et al. [19]. Many previous works compared this heuristic to many other heuristics and concluded that it produces the most efficient schedule with the minimum makespan [18, 21, 33–36]. The heuristic has a complexity of $O(t^2 p)$ where t represents the number of tasks to be executed and p represents the number of processors that execute the tasks. $O(t^2 p)$ is a low complexity compared to other proposed heuristics. Like most list-scheduling algorithms, the HEFT algorithm schedules tasks by running through two phases. The first phase is the *prioritization phase* where all tasks are ranked according to each task's execution priority. The ranking function is usually a function of the task weight and the communication costs between tasks. The HEFT algorithm defines rank by

$$rankHEFT(n_i) = \overline{wi} + \max_{n_j \in succ(n_i)} \{ \overline{ci, j} + rankHEFT(n_j) \}$$

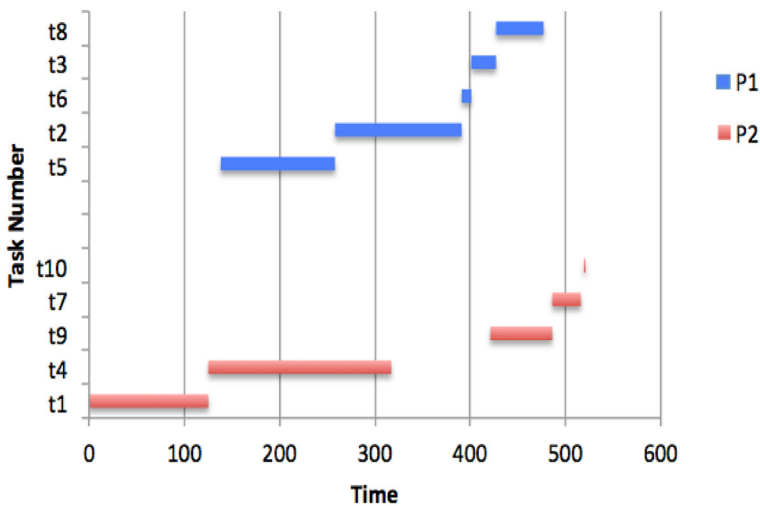
The ranking function in $rankHEFT(n_i)$ represents the length of the longest path between the current node and the exit node in the DAG. The ranking function is a recursive function that starts from the exit node of the graph recursively to end with the starting task in the DAG. The exit node's rank is calculated by a set of processors' average computation time. Referring to same ranking equation, \overline{wi} denotes the average computation time to execute task n_i by the set of processors. Additionally, $\overline{ci, j}$ denotes the communication cost between task n_i and task n_j . In the end, the task list is organized in a decreasing order such that tasks with higher value have a higher execution priority.

The second phase of the algorithm is the *processor selection* phase where each task on the ranked list is linked to either of the processors to be executed on. The processor that makes the earliest finish time of task n_i is selected. However, when possible, the HEFT algorithm inserts tasks at the earliest possible time between two already scheduled tasks on one processor.

Since we are going to compare the results of the proposed algorithm with HEFT [19], we illustrate its details with an example. Referring to the example in Fig. 1, we calculated the ranking values of $rankHEFT(n_i)$ and presented them in Table 1. The HEFT algorithm ranks the tasks in a decreasing order according to their priority as {t1, t5, t4, t2, t6, t3, t9, t8, t7, t10}. Next, each task in the priority list is assigned to the best processor to be executed. In the HEFT algorithm, the processor that executes the task with the minimum earliest finish time (EFT) is selected. For example, it is assumed that all processors are available. Therefore, earliest start time (EST) is equal to zero for both processors in the example. Then, the EFT is calculated for both processors. Processor 1 can execute t1 in 171 time units while processor 2 can execute it in 125 time units. Therefore, since processor 2 gives the minimum EFT, it is selected to execute t1. When t1 is completed, the next task, t5, starts execution according to its selected processor. This process is continued until the last task, t10, is completed. As for our example, for HEFT algorithm, all tasks complete their execution with a makespan of 521 time units. Furthermore, Table 1 details the scheduling data produced by the

Table 1 Schedule produced by the HEFT algorithm

Tasks	$rank_{HEFT}(ni)$	Processor selected	Earliest start time (EST)	Earliest finish time (EFT)
t1	507.5	2	0	125
t5	346.5	1	138	258
t4	313.0	2	125	317
t2	291.0	1	258	391
t6	218.5	1	391	401
t3	178.0	1	401	427
t9	137.5	2	421	486
t8	132.5	1	427	477
t7	83.5	2	486	516
t10	2.5	2	519	521

**Fig. 2** Example of the HEFT scheduling algorithm

HEFT algorithm. Figure 2 presents the detailed schedule for the same example. As shown in the figure, tasks {t1, t4, t9, t7, and t10} are executed on processor 2 where tasks {t5, t2, t6, t3, and t8} are executed on the other processor.

Another well-known list-scheduling algorithm is the Predict Earliest Finish Time (PEFT) algorithm proposed by Arabnejad et al. [14] which has the same complexity as HEFT, $O(t^2p)$. The PEFT introduces a look-ahead feature by computing an Optimistic Cost Table (OCT). The values introduced in OCT tables are optimistic costs where a processor's availability is not considered [14]. As in the HEFT algorithm, the PEFT has two main phases: a task prioritizing phase and a processor selection phase.

Table 2 Schedule produced by the PEFT algorithm

Tasks	$rank_{OCT}(ni)$	Processor selected	EST	EFT	OEFT
t1	247.5	2	0	125	376
t5	95.5	2	125	309	376
t2	82.0	1	142	275	372
t4	70.5	1	275	420	494
t6	55.5	1	420	430	483
t3	40.0	2	309	440	472
t9	2.5	2	440	505	507
t8	2.5	1	430	480	483
t7	2.5	2	505	535	537
t10	0.0	2	535	537	537

The PEFT algorithm starts by calculating the optimistic cost table on which the task ranking is based. The OCT value of a task n_i on processor p_k is defined by

$$OCT(ni, pk) = \max_{tj \in succ(ti)} \left[\min_{pw \in P} \{OCT(nj, pw) + w(nj, pw) + \overline{ci, j}\} \right],$$

$$\overline{ci, j} = 0 \quad \text{if } pw = pk$$

The recursive feature of the OCT calculates the maximum time of the shortest path between the current task and the exit node. All processors that are considered in the processor selection phase should be included in the calculations of the table. In the same equation, $w(nj, pw)$ denotes the execution time of task nj on processor pw . Moreover, $\overline{ci, j}$ denotes the average communication cost between tasks. As for the exit node, $OCT(nexit, pw) = 0$ for all processors. After computing all OCT values for all possible combinations between tasks and processors, the task ranking is calculated. In the PEFT, task ranking is computed by taking the average OCT for each task as defined by formula.

$$rank_{oct}(ni) = \frac{\sum_{k=1}^P OCT(ni, pk)}{P}$$

Referring to the example in Fig. 1, $rank_{oct}$ is calculated to get the values in Table 2. The PEFT ranks the task priority in decreasing order to get the following prioritization list {t1, t5, t2, t4, t6, t3, t9, t8, t7, t10}. Compared to the HEFT, tasks t2 and t4 have swapped their order of execution.

As for the second phase of the PEFT, the processor selection phase, the Optimistic EFT (O_{EFT}) is calculated. All detailed values are provided in Table 2. O_{EFT} is the sum of EFT and the computation time of the longest path from the current node to the exit node. Therefore, instead of selecting the processor that gives the earliest finish time, a looking forward method is achieved by selecting the shortest finish time for the tasks in the future. Therefore, O_{EFT} is defined by

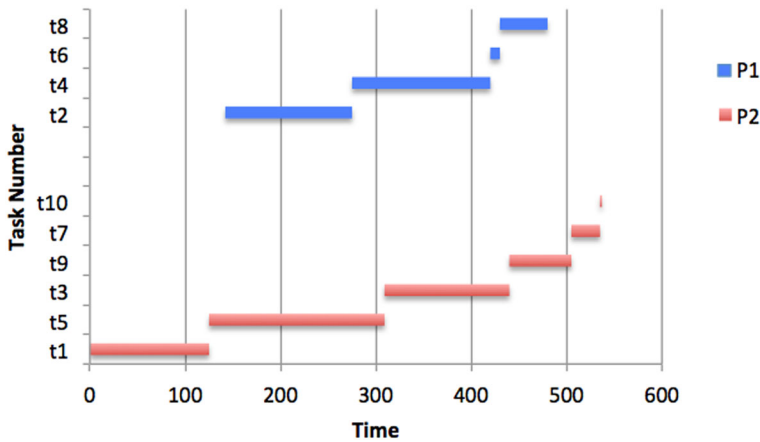


Fig. 3 Example of the PEFT scheduling algorithm

$$OEFT(ni, pj) = EFT(ni, pj) + OCT(ni, pj)$$

Following the PEFT algorithm to schedule the tasks for the example in Fig. 1 leads to a makespan of 537 time units. Figure 3 presents the details of the schedule for the same example.

As shown in Fig. 3, scheduling the tasks of the same example following the PEFT algorithm leads to the execution of tasks {t1, t5, t3, t9, t7 and t10} on processor 2 where tasks {t2, t4, t6 and t8} were executed on processor 1. Therefore, the total makespan to execute the tasks by the PEFT scheduling algorithm was 537 time units.

Hagras et al. [27] proposed the Heterogeneous Critical Parent Trees (HCPT) algorithm where a list (L) is used instead of prioritizing tasks. HCPT groups tasks into sets of “unlisted-parent trees.” A critical node is defined as the node that has equal Average Earliest Start Time (AEST) and Average Latest Start Time (ALST). Therefore, the algorithm works by executing two phases. The first phase is listing tasks and the second phase is assignment of the tasks to the processors to be executed. In the listing tasks phase, the HCPT starts with an empty list (L) and a stack (S) that contains critical nodes in decreasing order of their ALSTs. Next, if the top node in S has a parent node that is not in the list (unlisted-parent), then this parent is pushed to be executed. In the second phase, the tasks are assigned to the processors that allow earliest execution time.

Ilavarasan et al. [28] introduced the High-Performance Scheduling (HPS) algorithm. The HPS includes three phases: level sorting, task prioritization and processor selection. In the first phase, the input application DAG is traversed in a top-to-bottom way to sort the tasks level by level. Tasks that are independent of each other are assigned to different groups. Therefore, it is possible to execute tasks in the same level in the DAG in parallel. In the second phase, the tasks are prioritized according to Down Link Cost (DLC), Up Link Cost (ULC) and Link Cost (LC) of the tasks. The DLC is the highest communication cost among all parent nodes (tasks). The ULC is the highest communication cost among all of the tasks’ child nodes. The LC is the sum of the two

previous values and the highest LC of all immediate parent nodes in the DAG. Next, based on the LC values, the tasks are prioritized by decreasing LC values. Therefore, tasks with a higher LC value are executed first. In the processors selection phase, the processor that gives the lowest earliest finish time is selected.

Ilavarasan et al. [29] proposed the Performance Effective Task-Scheduling (PETS) algorithm that has similar phases to the HPS algorithm. Like HPS, the tasks are categorized according to their levels. Next, in the prioritization phase, the priorities are calculated according to the Average Computation Cost (ACC), Data Transfer Cost (DTC) and the Rank of Predecessor Task (RPT). The ACC is the average of computation costs on all the sets of processors. The DTC is the amount of communication costs between a task and its successor. The RPT is the highest rank of all immediate parent nodes. Therefore, the rank in the prioritization phase is calculated by adding the values of ACC, DTC and RPT. Then, the tasks are ranked from the highest total value to the lowest. Like most of the previously proposed algorithms, the processor selection is conducted by selecting the processor that has the minimum EFT.

Dai et al. [16] proposed the HCPPEFT algorithm that uses three levels of priority in the algorithm to choose the task. First level is for critical tasks, second level is for tasks with longer path to exit node and the third level is for tasks with less number of predecessors. However, in the processor selection task duplication is adopted to minimize the schedule length.

A recent study conducted by Shetti et al. [18] introduced a non-cross mechanism to enhance the CPU–GPU environment. The authors attempted to optimize the processor selection by reducing the cross over between the CPU and the GPU processors. In addition, the authors proposed a composite ranking function which takes into consideration both the ratio of the execution time on the slowest processor to the faster processor and the absolute time difference of the computation times among processors. By having the same complexity of previous algorithms, $O(t^2 p)$, the proposed algorithm showed overall makespan reduction. However, this recent study did not consider communication time between tasks to determine the ranking of tasks. We are extending Shetti et al.'s [18] work in the proposed technique by incorporating the communication time between tasks for priority determination. Similar to HEFT, the proposed algorithm had two phases: a prioritization phase and a processor selection phase. As for the prioritization phase, the weight value that contributed to calculating the rank's recursive equation was enhanced by considering both local and global optimal time values. This mechanism aided in prioritizing tasks without giving the larger tasks higher priorities than smaller tasks. In the processor selection phase, instead of using a fixed value for threshold to make the decision to cross over between one processor to another as suggested by Shetti et al. [18], we proposed a randomized range to better explore the solution space. This mechanism proved to produce better results than applying previously proposed algorithms to the same problem. In this study, we benefit from the cross over mechanism to enhance the proposed scheduling algorithm.

4 The proposed algorithm

In this section, we propose a new technique to solve the problem of list-based scheduling of tasks. Like all previously suggested algorithms, our proposed algorithm's main goal is to minimize the overall execution makespan by taking into consideration all data transfers and dependencies between tasks. Moreover, our algorithm considers the communication costs and processing time of each processor. In this section, we will start by presenting the details of the proposed algorithm. Then, we will present an illustrative example to show the capabilities of the proposed algorithm and compare its results with HEFT [19] and PEFT [14] algorithms.

4.1 Description of the proposed algorithm

The proposed algorithm starts with the *prioritization phase* where the tasks are ranked according to their priority to be executed by the set of processors. Next, according to the prioritization phase output and for each task, the processor that most efficiently executes the task will be selected in the *processor selection phase* [21]. We are extending Shetti et al.'s [18] work in the proposed technique by incorporating the communication time between tasks for enhancing prioritization and process selection. Tasks are ranked according to a calculated weight of execution time of processors, communication costs between tasks and the prioritization value of the previous task.

Regarding the processor selection phase, the algorithm introduces a randomized decision based on a threshold value to cross over among processors. In addition, we take into consideration the communication costs between each task to compute the threshold.

4.1.1 Prioritization phase

The proposed algorithm starts by calculating $Weight_{ni}$ to be considered in the prioritization calculations. $Weight_{ni}$ is calculated for each task in the graph input. $Weight_{ni}$ is the absolute value of the ratio of time difference between the highest execution time and the lowest execution time and the speedup of the considered set of processors. In HEFT [19] and PEFT [14] algorithms, the computation time is calculated by retrieving the average between times of execution on each processor. In contrast, the computation time in our proposed algorithm is the same as defined by Shetti et al. [18], which is defined by taking the absolute value of the time difference between the execution times over the speedup. Therefore, the computation time is defined by

$$Weight_{ni} = \left| \frac{w(ni, pj) - w(ni, pk)}{w(ni, pj) / w(ni, pk)} \right|$$

where $w(ni, pj)$ is the execution time of task on processor pj . There exist several choices to assign a single value corresponding to varying task execution times on heterogeneous computing systems such as the average, median, maximum or minimum values of execution cost to better rank the tasks. But none of these metrics showed

consistent performance [36]. However, $Weight_{ni}$ captures more information about the tasks for ranking.

To complete the *prioritization phase* of the scheduling algorithm, the calculated $Weight_{ni}$ is then substituted in the $rank_{proposed}(ni)$ to get the task ranking. This substitution considers that, like all previous scheduling algorithms, the input graph is traversed in a bottom-to-top topology.

$$rank_{proposed}(ni) = Weight_{ni} + \max_{nj \in succ(ni)} \{ \overline{ci}, j + rank_{proposed}(nj) \}$$

4.1.2 Processor selection phase

The selection phase is enhanced by non-crossover between processors as proposed by Shetti et al. [18]. However, instead of using a fixed value for a threshold, we make randomized decision within a certain range of values. Unlike HEFT and PEFT, the non-crossover technique does not select the processor that produces the lowest finish time. Instead, once the first processor is selected to execute the first task, the cross over to another processor depends on a calculated value called Cross-Threshold, which can be any number between 0 and 1. The closer the value is to 1, the more the processors will cross over, and this will lead to have an algorithm that is similar to HEFT. On the other hand, decreasing the value of the cross-threshold does not allow cross over to occur easily. Note that cross over means assigning the task to the processor with the lowest execution time (local optimal) as compared to the processor that produces the lowest finish time (globally optimal). Since sometimes, better schedule length can be generated by making a local optimal decision as compared with the global optimal decision. Therefore, the algorithm uses a balanced combination of a global optimal and local optimal decisions to explore the search space efficiently and efficiently to find the minimum makespan schedule. The cross over is not related to task migration where a task starts its execution in one processor, but then the task is interrupted and moved to another processor to continue its execution in case of preemptive scheduling algorithms. In this paper, we are dealing with non-preemptive scheduling techniques and tasks do not get migrated from one processor to another during execution.

Algorithm 1 formally describes the process which follows Shetti et al. [18] style. First, in lines 1–3, $rank(n_i)$ is calculated considering the modified computation of weight (n_i). The tasks are then saved in a ready-list queue that works in a first-in, first-out mechanism according to the priorities given in the previous step. Next, while the ready-list queue is not empty, each task is assigned to a selected processor. Therefore, going through a decreasing rank, lines 4 through 8 compute the earliest start time and earliest finish time by taking into consideration the communication time and execution time of each processor. It should be noted that communication time between two tasks is added to the earliest start time in the cases where the parent processor was executed by a different processor than the current processor that is calculating the EST. Next, the processor with the minimum EFT time is selected to be compared with the minimum processor weight w_i, j . Furthermore, if the case in line 10 is true, then both global and local results are equal. Therefore, in this case, the same processor with the lowest global and optimal result is chosen, and the available time of the processor and the

ready-list are updated accordingly. In the case that line 10 is not true, another processor exists that has a lower computation time than the first processor. Therefore, in this case we calculate another value of $Weight_{abstract}$ which is the absolute value of the saved time over the speedup as shown in the following equation:

Algorithm 1: Proposed Algorithm

begin

```

1.  for all  $n_i$  in  $N$  do
2.      Compute  $rank_{proposed}(n_i)$ ;
3.  end
4.  while ready-list is NOT Empty do
5.       $n_i \leftarrow$  task in the ready-list with the maximum rank;
6.      for all  $p_j$  in  $P$  do
7.          Compute  $EFT(n_i, p_j) \leftarrow w_{i,j} + EST(n_i, p_j)$ ;
8.      end
9.      Select  $p_j$  with Min  $EFT(n_i, p_j)$ ;
10.     if  $w_{i,j} \leq \text{Min } k \in P(w_{i,k})$  then
11.         Assign  $n_i$  to the processor  $p_j$  that minimize  $EFT$  of task  $n_i$ ;
12.         Update  $T_{available}(p_j)$  and ready-list;
13.     else
14.         Compute  $Weight_{abstract}$  and  $Cross\_Threshold$ ;
15.         if  $Cross\_Threshold \leq r$  then           //  $r$  is a random number
16.             Map node  $n_i$  to processor  $p_j$ ;       // cross over
17.             Update  $T_{available}(p_j)$  and ready-list;
18.         else
19.             Map node  $n_i$  to processor  $p_k$ ;       // no cross over
20.             Update  $T_{available}(p_k)$  and ready-list;
21.         end
22.     end
23. end

```

$$Weight_{abstract} = \left| \frac{EFT(n_i, p_j) - EFT(n_i, p_k)}{EFT(n_i, p_j) / EFT(n_i, p_k)} \right|$$

Next, the value of the $Cross_Threshold$ is computed based on the following equation:

$$Cross_Threshold = \frac{Weight_{ni}}{Weight_{abstract}}$$

If the value of $Cross_Threshold$ is less than or equal to a randomly generated number (r) within the range of 0.1–0.3, the task is crossed over to the second processor. This step makes randomized decision to explore the solution space effectively. On the other hand, if the value of $Cross_Threshold$ is larger than the random value, no cross over should be allowed. Therefore, the task will be executed on the same processor that was chosen previously.

4.2 Illustrative example

When applying our proposed algorithm to the same problem in Fig. 1, we first started by calculating the $rank_{proposed}(ni)$ for each task recursively, starting from t10 and ending at t1. The details of the values are presented in Table 3. The output of applying the proposed ranking mechanism lead to a prioritization queue of {t1, t5, t4, t2, t6, t8, t3, t9, t7, t10}. When compared with the HEFT and PEFT prioritization techniques as explained in Sect. 2, we found that the ranking labeled task 8 a higher priority than the HEFT and PEFT algorithms. This case happened because our proposed algorithm does not only calculate the average computation time, but it also neglects to highly prioritize larger tasks. Instead, a new calculation of time over speedup is considered to determine the weight.

Moving to the next phase, both processors were available at the start, so the EST of t1 on both processors was equal to zero. $EFT(t1, P1) = 171$ and $EFT(t1, P2) = 125$. Since processor 2 had the minimal EFT, the processor was selected to execute t1. As shown in Fig. 4, the processor executed P2 from time zero to 125. Hence, all other tasks could begin after t1 completed execution.

Next, $EST(t5, P1)$ and $EST(t5, P2)$ calculated values of 138 and 125 simultaneously. Therefore, $EFT(t5, P1) = 258$ and $EFT(t5, P2) = 309$. Since P1 was the most efficient processor for t5 (the processor that executed the task in the least time) and the processor that produced the least EFT, P1 was the chosen processor for t5. Figure 5 shows the details of scheduling t5.

Next, $EST(t4, P1)$ and $EST(t4, P2)$ calculated values of 258 and 125 simultaneously. Therefore, $EFT(t4, P1) = 403$ and $EFT(t4, P2) = 317$. Since P1 produced the lowest execution time and P2 produced the lowest EFT, we evaluated whether cross over from P1 to P2 should be applied. $Weight_{ni}/Weight_{abstract}$ was calculated and compared with the cross threshold assumed previously. For this step in the example, $Weight_{ni}/Weight_{abstract}$ is equal to $\frac{35.5}{\left(\frac{403-317}{317}\right)} = 0.5$. Since 0.5 is greater than 0.3

Table 3 Schedule produced by the proposed algorithm

Tasks	rankproposedt (ni)	Processor selected	Earliest start time (EST)	Earliest finish time (EFT)
t1	195.5	2	0	125
t5	149.0	1	138	258
t4	119.5	1	258	403
t2	97.0	2	125	239
t6	87.0	1	403	413
t8	73.0	1	413	463
t3	68.5	2	238	370
t9	50.5	2	410	475
t7	32	2	475	505
t10	0.5	2	505	507

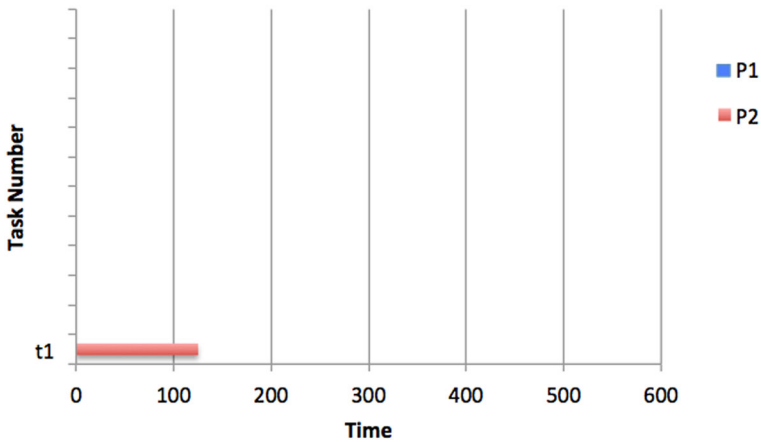


Fig. 4 Scheduling t1 to processor 2

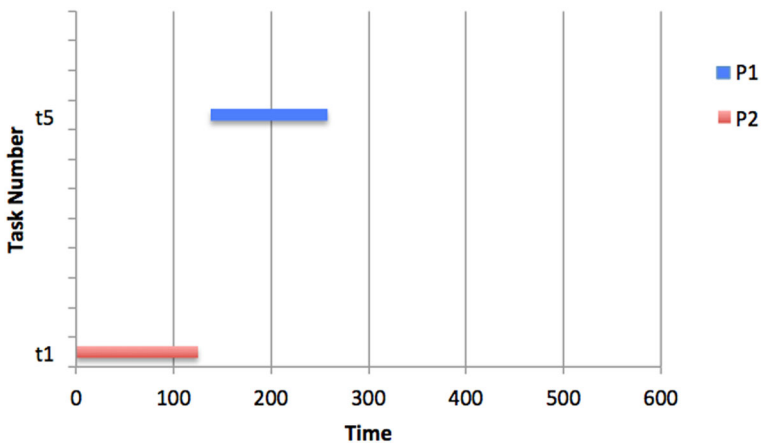


Fig. 5 Scheduling t5 to processor 1

(assumed random number $r = 0.3$) no cross over was allowed. Therefore, as shown in Fig. 6, t4 was executed on the processor that produced the lowest execution time (P1) not taking into consideration that P2 produced the overall lowest EFT.

The algorithm continued using the same procedure for t2, t6, and t8. Next, the t3 schedule applied the cross over from the processor that produced the lowest execution time (P1) to the processor that produced the lowest EFT (P2). To detail the cross over step, we started with $EST(t3, P1) = 463$ and $EST(t3, P2) = 239$. Therefore, $EFT(t3, P1) = 489$ and $EFT(t3, P2) = 370$. Since the processor that produced the lowest execution time was different than the processor that produced the lowest EFT, $Weight_{ni}/Weight_{abstract}$ should be calculated. In this case, $Weight_{ni}/Weight_{abstract} = 0.23$. Since the value of cross threshold is 0.23, which is less than the random value (assumed random number $r = 0.3$), cross over from P1 to P2 was applied and, therefore, t3 was executed on P2 as shown in Fig. 7.

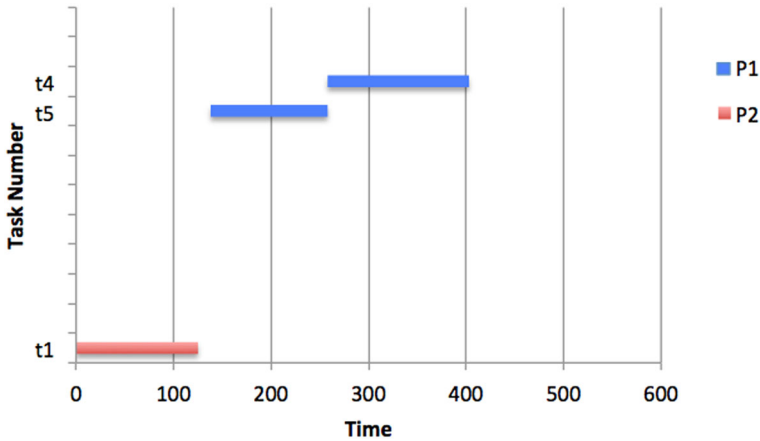


Fig. 6 Scheduling t4 to processor 1

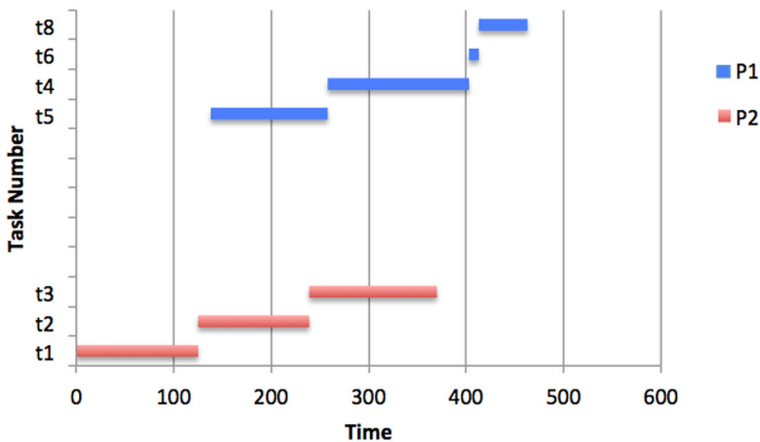


Fig. 7 Scheduling t3 to processor 2

Next, following the same procedure t9, t7, and t10 were executed on P2 to produce the schedule shown in Fig. 8.

Due to the processor selection mechanism of limited cross over between processors, the algorithm schedules the tasks with a makespan of 507. Therefore, as shown in Table 3, tasks {t1, t2, t3, t9, t7, and t10} were executed on processor 2 while tasks {t5, t4, t6, and t8} were executed on processor 1.

In summary, the limited cross over mechanism produced better results in the processor selection phase, especially for applications with larger sets of tasks. Comparing the HEFT, the PEFT and the limited cross over proposed algorithm resulted in a makespan of 521, 537 and 507, respectively.

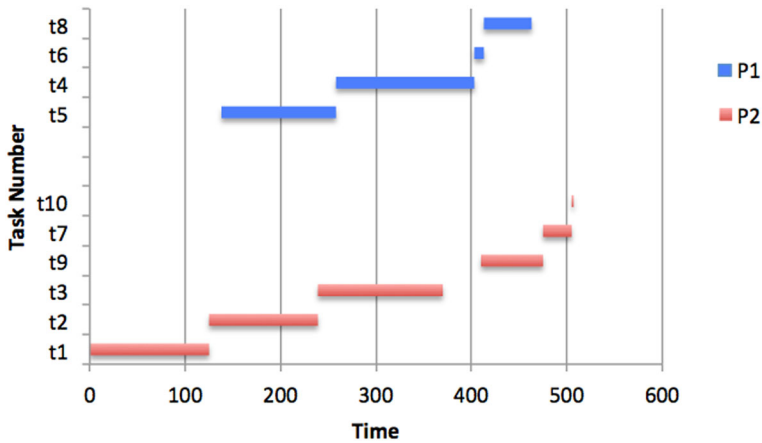


Fig. 8 Example of the proposed scheduling algorithm

4.3 Complexity

List-based scheduling algorithms depend on the number of processors and the number of tasks that are in the input application. Moreover, the HEFT, the PEFT and our proposed algorithm have similar basic phases for scheduling. The algorithm first prioritizes the tasks and assigns a priority rank to each task. Then, the algorithm schedules the tasks using one while loop and one for loop to run the schedule on each task. Therefore, our algorithm, like HEFT and PEFT, has a complexity of $O(t^2 p)$, where t is the number of tasks in the input graph and p is the number of processors in the set of processors considered.

5 Experimental results and analysis

In this section, we report the comparative evaluation of the proposed algorithm with two well-known scheduling algorithms, the HEFT [19], and the PEFT [14] using various performance metrics. We first describe the comparison metrics used to evaluate the performance of different algorithms. Next, the simulation environment, including the algorithm implementation and generation of random directed acyclic graphs (DAGs), is described. Finally, the performance results and analysis are presented.

5.1 Comparison metrics

Algorithms are compared based on the following performance metrics:

A. Makespan

The most used comparison metric for a single DAG is the makespan. The makespan is the time where the last task in the input DAG completes its execution. In other words,

the makespan of an algorithm for a DAG is the actual time for the exit node to be executed. The makespan of an algorithm can be defined by

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

where $AFT(n_{exit})$ represents the Actual Finish Time of the exit node of the input graph. If there are more than one exit nodes in the graph, the maximum finish time among all exit nodes is considered as the makespan.

B. Scheduling length ratio (SLR)

The Scheduling Length Ratio (SLR) represents the makespan normalized to the lower bound [21]. The SLR is defined by

$$SLR = \frac{makespan(solution)}{Critical\ Path\ Including\ Communication\ (CPIC)}$$

The Critical Path Including Communication (CPIC) is the minimum cost of the critical path tasks including communication costs [18]. Since there is no makespan less than the CPIC, when an SLR is lower, the algorithm is better.

C. % Improvement in schedule length

We evaluate our proposed algorithm in terms of percentage improvement in schedule length (SL) as compared with the existing approaches. The percentage improvement in schedule length is defined as:

$$\% \text{ SL}_{improvement} = \left(1 - \frac{SL_{Pr}}{SL_A}\right) \times 100$$

where SL_{Pr} represents the schedule length generated by the proposed algorithm and SL_A is the schedule length generated either by HEFT or PEFT algorithms. A higher value for percentage improvement means that the proposed algorithm generated the shortest length schedule as compared with other algorithms.

5.2 Simulation setup

The proposed algorithm and the two well-known scheduling algorithms HEFT and PEFT were implemented in Java using Eclipse SDK (version 4.2.0) on a MacBook Pro with OS X (version 10.8.5), 2.7 GHz Intel Core i7 processor, and 4G of memory. Many different random graphs were generated using the Directed Acyclic Graph Generator [33] to evaluate the performance of each algorithm. Some modifications to the graph generator program were made to generate random data for input into the proposed algorithm.

The modified DAG Generator considers many different parameters to generate random graphs with different characteristics. Some of the main parameters are: number

of nodes in the graph, width of the graph, density of the edges between two levels in the graph, and the number of edges that a task requires to jump from one level to another level. Using the DAG Generator, 750 random graphs were generated and tested as input into the proposed algorithm. To generate different graphs with different characteristics, the following parameters were set and combined in all possible combinations:

- Number of tasks (n) = {10, 50, 100, 200, 500}
- Graph shape (FAT) = {0.1, 0.5, 1, 5, 10}
- Computation to Communication Ratio (CCR) = {0.1, 5, 10}
- Density = {0.5}
- Jump = {5}

In these parameters, the number of tasks (n) is the number of nodes in a DAG input. FAT is the parameter that affects the height and width of the DAG. A small value for the FAT parameter would lead to a thin DAG, whereas a larger value leads to a wider DAG with a higher degree of children per node. To generate computation and communication costs with a range of values, the Computation to Communication Ratio (CCR) was changed over the randomly generated graphs. CCR is the ratio of the sum of edge weights to the sum of the node weights in a DAG. Furthermore, the density of a generated DAG is known as the number of edges between two levels of the input graph. A large value of density leads to large number of edges, while a smaller value leads to a smaller number of edges. Finally, jump represents that an edge can go from level l to a level $l + \text{jump}$. Thus, having a high value for jump parameter provided the chance of having children nodes from higher levels. For example, if jump was equal to 5, then the starting node could have its direct children anywhere from level 2 to level 6.

5.3 Performance results and analysis

In this section, we present details of the performance results and analysis of the proposed algorithm across a wide range of randomly generated DAGs by comparing against the two well-established scheduling algorithms, the HEFT algorithm and the PEFT algorithm.

A. % Improvement in schedule length comparison

The percentage improvement in schedule length of the proposed algorithm was compared with the HEFT and the PEFT algorithms across DAGs with different parameters. The results show that, on average, for fewer nodes, the HEFT and the PEFT achieve a lower makespan when compared with the proposed algorithm. However, as the number of nodes gets larger, the results of the proposed algorithm are better. Figures 9, 10 and 11 show the percentage improvement in schedule length as compared with HEFT algorithm across different graph shapes and different numbers of nodes. Figure 9 shows that for fewer nodes and with a large value of CCR, HEFT achieves the shortest length schedule. However, for larger number of nodes, the proposed algorithm achieves a maximum of 6% reduction in the schedule length as compared with HEFT

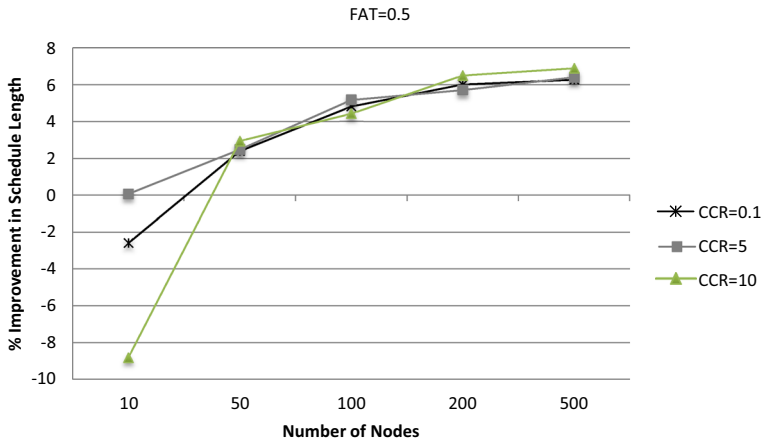


Fig. 9 % Improvement in schedule length as compared to the HEFT algorithm with $FAT = 0.5$

algorithm. A similar behavior is observed in Fig. 10. However, for 100 nodes, the proposed algorithm shows a 13% reduction in schedule length as compared to HEFT algorithm. As the FAT of a graph increases, the improvement in schedule length drops to 3% as shown in Fig. 11. In general, the results show that the proposed algorithm works better for graphs with larger nodes and with graphs that have a narrower shape as compared with HEFT. For narrower graphs, the speedup reaches up to 6–7%. Overall, the proposed algorithm gives better results than the HEFT algorithm for 83% of the randomly generated DAGs.

Furthermore, as is shown in Figs. 12, 13 and 14, when comparing the proposed algorithm to the PEFT algorithm, improvement in schedule length reaches to 7%, especially for graphs with a narrower shape. Even for fewer nodes, in some cases, the proposed algorithm achieves an improvement of 9% in schedule length as shown in Fig. 14. Overall, our proposed algorithm gives better results than the PEFT algorithm for 84% of the randomly generated DAGs.

B. Scheduling length ratio comparison (SLR)

The SLR across randomly generated graphs was calculated for each algorithm. The mean, median, standard deviation and best case values for SLR for the algorithms are shown in Table 4. All the values show an enhancement in the overall performance of the proposed algorithm against the HEFT algorithm. The results show that the improvement is consistently better than HEFT in comparison to graphs with different CCRs and different shapes. Moreover, it is observed that for the best case values across all of the DAGs, the proposed algorithm generated better results than the HEFT algorithm 80% of the time.

Figure 15 demonstrates the performance of algorithms across varying graph shapes by changing the value of FAT parameter. The results show that the proposed algorithm overall produces better results as compared with the HEFT and PEFT.

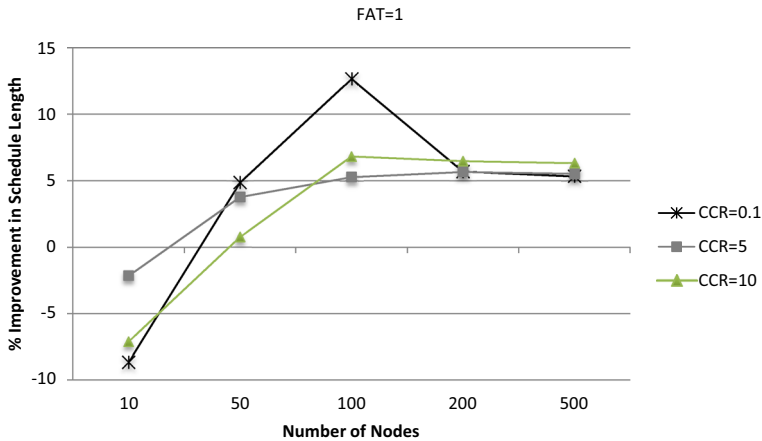


Fig. 10 % Improvement in schedule length as compared to the HEFT algorithm with $FAT = 1$

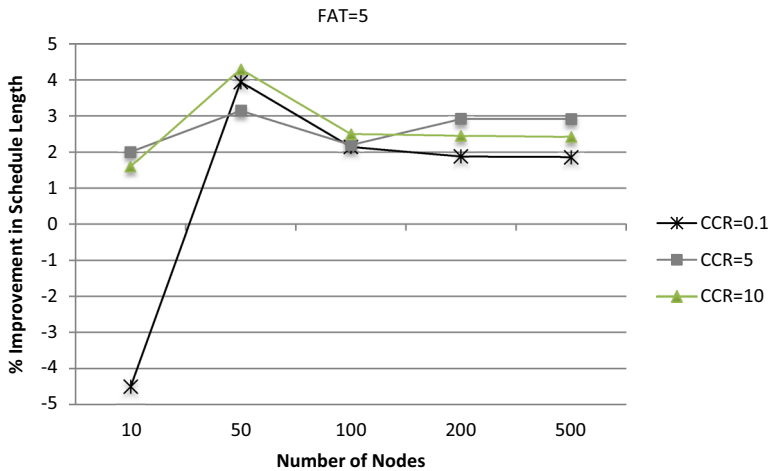


Fig. 11 % Improvement in schedule length as compared to the HEFT algorithm with $FAT = 5$

Another important observation is the comparison of SLR over varying values of CCR. Figure 16 shows a large improvement overall when scheduling tasks using the proposed algorithm. Furthermore, the results show that the proposed algorithm produces the best results for graphs with larger CCRs. This is due to the fact that the proposed algorithm limits the amount of cross over between processors because of the communication cost that must be added to allow transfer from one processor to another.

Overall, all experimental results show that the proposed algorithm obtains better results as compared with the two well-known algorithms, HEFT and PEFT. All comparison metrics show that the proposed approach has a major impact on large sets of graphs due to the enhancement in the prioritization and processor selection phases of the list-scheduling algorithm.

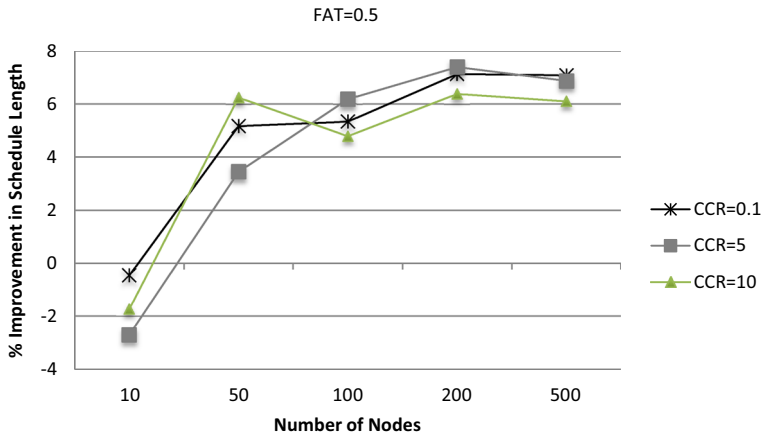


Fig. 12 % Improvement in schedule length as compared to the PEFT algorithm with FAT = 0.5

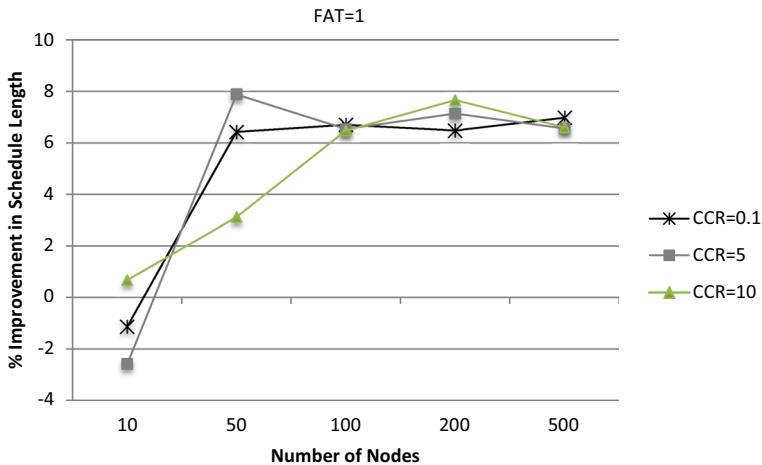


Fig. 13 % Improvement in schedule length as compared to the PEFT algorithm with FAT = 1

6 Conclusion and future work

In this paper, we have presented an enhancement to the list-based heterogeneous task-scheduling problem, in which tasks are assigned to the processors with the objective to minimize the overall makespan of execution. In the proposed algorithm, we introduced enhancements on both the task prioritization and the processor selection phases. As for the prioritization phase, the weight value that contributed to calculating the rank's recursive equation was enhanced by considering both local and global optimal time values in addition to incorporating the communication time between tasks. This mechanism helped to prioritize tasks without giving the larger tasks higher priorities than

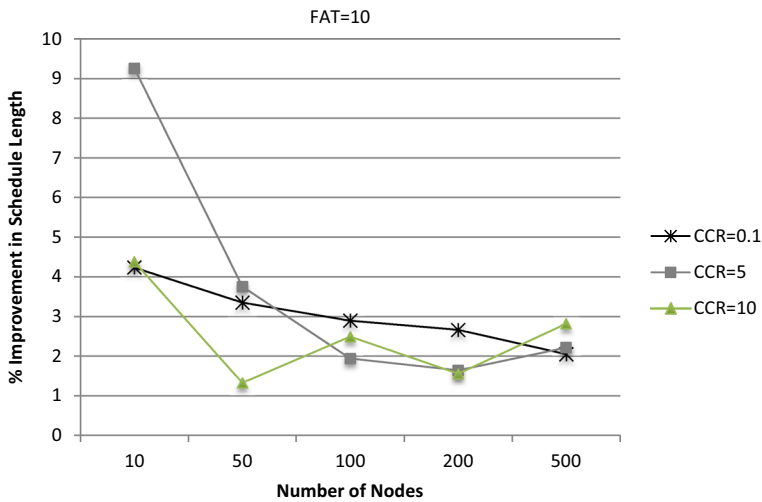


Fig. 14 % Improvement in schedule length as compared to the PEFT algorithm with FAT = 10

Table 4 SLR comparison over varying CCR

	CCR= 0.1			CCR= 5			CCR= 10		
	HEFT	PEFT	Proposed algorithm	HEFT	PEFT	Proposed algorithm	HEFT	PEFT	Proposed algorithm
Mean	6.80	6.92	6.42	7.05	7.11	6.46	7.51	7.8	6.87
Median	6.35	6.47	6.13	8.32	8.64	8.01	7.49	8.55	7.23
SD	5.30	5.21	5.07	4.37	4.33	3.90	5.17	4.82	4.77
Best case %	16.1	13.37	80.42	15.45	12.8	75.07	14.34	16.92	79.8

smaller tasks. As for the processor selection phase, the algorithm introduced a randomized decision making mechanism based on a threshold value to cross over among processors to reduce the overall execution makespan. We applied the proposed algorithm to 750 runs of random task graphs, and the results showed improvement in the makespan that reached up to 6–7 % as compared with the HEFT [19] and the PEFT [14], which are the two well-known algorithms for the heterogeneous processor scheduling problem. The proposed algorithm has the same complexity as that of the previously proposed algorithms, but it considerably improved the performance by minimizing the execution makespan.

A promising future direction for this research is to consider task duplication in the processor selection phase to further minimize the makespan by reducing communications between processors. Another extension to this study could include memory

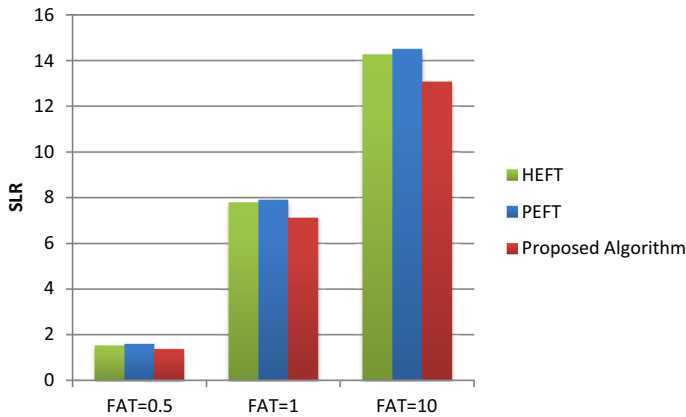


Fig. 15 SLR comparison over different graph shapes

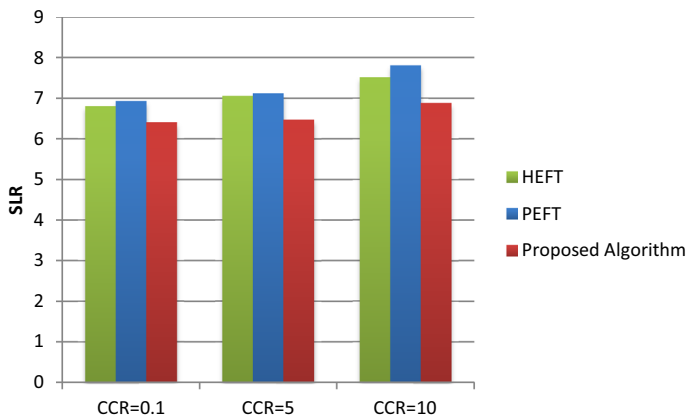


Fig. 16 SLR comparison over different CCR

constraints [33] or energy-aware data allocation and task scheduling with the goal of optimizing the total time to execute and to minimize the total system energy consumption [35].

Acknowledgements Thanks are extended to anonymous reviewers for their constructive comments which improved overall quality of the manuscript.

References

1. International Data Corporation, Worldwide and regional public cloud services 2012–2016 forecast. <http://www.idc.com>. Accessed 10 Dec 2015
2. Yang CH, Lee P, Chung YC (2007) Improving static task scheduling in heterogeneous and homogeneous computing systems. In: IEEE Parallel Processing, 2007. ICPP 2007. International Conference on, pp 45–45
3. Arabnia HR, Oliver MA (1987) Arbitrary rotation of raster images with SIMD machine architectures. Int J Eurograph Assoc (Computer Graphics Forum) 6(1):3–12

4. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. *Int J Eurograph Assoc (Computer Graphics Forum)* 8(1):3–11
5. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. *J Parallel Distrib Comput* 10(2):188–193
6. Bhandarkar SM, Arabnia HR, Smith JW (1995) A reconfigurable architecture for image processing and computer vision. *Int J Pattern Recognit Artif Intell (IJPRAI)* 9(2):201–229
7. Bhandarkar SM, Arabnia HR (1995) The Hough transform on a reconfigurable multi-ring network. *J Parallel Distrib Comput* 24(1):107–114
8. Bhandarkar SM, Arabnia HR (1995) The REFINE multiprocessor: theoretical properties and algorithms. *Parallel Comput* 21(11):1783–1806
9. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. *J Supercomput* 10(3):243–270
10. Wani MA, Arabnia HR (2003) Parallel edge-region-based segmentation algorithm targeted at reconfigurable multi-ring network. *J Supercomput* 25(1):43–63
11. Canon LC, Jeannot E, Sakellariou R, Zheng W (2008) Comparative evaluation of the robustness of dag scheduling heuristics. In: *Grid Computing*. Springer, US, pp 73–84
12. Young CP, Chang BR, Qiu ZL (2012) Scheduling optimization for vector graphics acceleration on multiprocessor systems. *J Inf Hiding Multimed Signal Process* 3(3):248–278
13. Zhou H, Liu C (2014) Task mapping in heterogeneous embedded systems for fast completion time. In: *ACM Proceedings of the 14th International Conference on Embedded Software*, pp 1–10
14. Arabnejad H, Barbosa JG (2014) List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Trans Parallel Distrib Syst* 25(3):682–694
15. Augonnet C, Thibault S, Namyst R, Wacrenier PA (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr Comput Pract Exp* 23(2):187–198
16. Dai Y, Zhang X (2014) A synthesized heuristic task scheduling algorithm. *Sci World J* 2014:1–9
17. Kuchcinski K (2003) Constraints-driven scheduling and resource assignment. *ACM Trans Design Automat Electron Syst (TODAES)* 8(3):355–383
18. Shetti KR, Fahmy SA, Bretschneider T (2013) Optimization of the HEFT Algorithm for a CPU-GPU Environment. In: *IEEE Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2013 International Conference on*, pp 212–218
19. Topcuoglu H, Hariri S, Wu MY (2002) Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans Parallel Distrib Syst* 13(3):260–274
20. Ullman JD (1975) NP-complete scheduling problems. *J Comput Syst Sci* 10(3):384–393
21. Arabnejad H (2013) List based task scheduling algorithms on heterogeneous systems-an overview. https://paginas.fe.up.pt/~prodei/ds12/papers/paper_30.pdf. Accessed 5 Jun 2016
22. Sakellariou R, Zhao H (2004) A hybrid heuristic for DAG scheduling on heterogeneous systems. In: *IEEE Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p 111
23. Kwok YK, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput Surv* 31(4):406–471
24. Boveiri HR (2015) An efficient task priority measurement for list-scheduling in multiprocessor environments. *Int J Softw Eng Appl* 9(5):233–246
25. N'Takpé T, Suter F, Casanova H (2007) A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms. In: *IEEE Parallel and Distributed Computing, 2007. ISPDC'07. Sixth International Symposium on*, pp 35–35
26. Venugopalan S, Sinnen O (2012) Optimal linear programming solutions for multiprocessor scheduling with communication delays. In: *Algorithms and architectures for parallel processing*. Springer, Berlin Heidelberg, pp 129–138
27. Hagrais T, Janecek J (2003) A simple scheduling heuristic for heterogeneous computing environments. In: *IEEE International Symposium on Parallel and Distributed Computing*, pp 104–110
28. Ilavarasan E, Thambidurai P, Mahilmanan R (2005) High performance task scheduling algorithm for heterogeneous computing system. In: *Distributed and Parallel Computing*. Springer Berlin Heidelberg, pp 193–203
29. Ilavarasan E, Thambidurai P (2007) Low complexity performance effective task scheduling algorithm for heterogeneous computing environments. *J Comput Sci* 3(2):94–103
30. Bittencourt LF, Sakellariou R, Madeira ER (2010) Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In: *IEEE In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pp 27–34

31. Falzon G, Li M (2012) Enhancing list scheduling heuristics for dependent job scheduling in grid computing environments. *J Supercomput* 59(1):104–130
32. Dhodhi MK, Ahmad I, Yatama A, Ahmad I (2002) An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems. *J Parallel Distrib Comput* 62(9):1338–1361
33. Herrmann J, Marchal L, Robert Y (2014) Memory-aware list scheduling for hybrid platforms. In: 2014 IEEE International In Parallel & Distributed Processing Symposium Workshops (IPDPSW), pp 689–698. doi:[10.1109/IPDPSW.2014.80](https://doi.org/10.1109/IPDPSW.2014.80)
34. Su S, Li J, Huang Q, Huang X, Shuang K, Wang J (2013) Cost-efficient task scheduling for executing large programs in the cloud. *Parallel Comput* 39(4):177–188
35. Wang Y, Li K, Chen H, He L, Li K (2014) Energy-aware data allocation and task scheduling on heterogeneous multiprocessor systems with time constraints. *IEEE Trans Emerg Topics Comput* 2(2):134–148
36. Huang KC, Tsai YL, Liu HC (2015) Task ranking and allocation in list-based workflow scheduling on parallel computing platform. *J Supercomput* 71(1):217–240

