List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table

Hamid Arabnejad and Jorge G. Barbosa, Member, IEEE

Abstract—Efficient application scheduling algorithms are important for obtaining high performance in heterogeneous computing systems. In this paper, we present a novel list-based scheduling algorithm called Predict Earliest Finish Time (PEFT) for heterogeneous computing systems. The algorithm has the same time complexity as the state-of-the-art algorithm for the same purpose, that is, $O(v^2.p)$ for v tasks and p processors, but offers significant makespan improvements by introducing a look-ahead feature without increasing the time complexity associated with computation of an Optimistic Cost Table (OCT). The calculated value is an optimistic cost because processor availability is not considered in the computation. Our algorithm is only based on an OCT table that is used to rank tasks and for processor selection. The analysis and experiments based on randomly generated graphs with various characteristics and graphs of real-world applications show that the PEFT algorithm outperforms the state-of-the-art list-based algorithms for heterogeneous systems in terms of schedule length ratio, efficiency and frequency of best results.

Index Terms—Application scheduling, DAG scheduling, random graphs generator, static scheduling.

1 Introduction

A Heterogeneous system can be defined as a range of different system resources, which can be local or geographically distributed, that are utilized to execute computationally intensive applications. The efficiency of executing parallel applications on heterogeneous systems critically depends on the methods used to schedule the tasks of a parallel application. The objective is to minimize the overall completion time or *makespan*. The task scheduling problem for heterogeneous systems is more complex than that for homogeneous computing systems because of the different execution rates among processors and possibly different communication rates among different processors. A popular representation of an application is the Directed Acyclic Graph (DAG), which includes the characteristics of an application program such as the execution time of tasks, the data size to communicate between tasks and task dependencies. The DAG scheduling problem has been shown to be NP-complete [14], [18], [31], even for the homogeneous case; therefore, research efforts in this field have been mainly focused on obtaining low-complexity heuristics that produce good schedules. In the literature, one of the best list-based heuristics is the Heterogeneous Earliest-Finish-Time (HEFT) [29]. In [11], the authors compared 20 scheduling heuristics and concluded that on average, for random graphs, HEFT is the best one in terms of robustness and schedule length.

• *Universidade do Porto, Faculdade de Engenharia, Dep. de Engenharia Informática, Laboratório de Intelegência Artificial e Ciência dos Computadores, Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal hamid.arabnejad@fe.up.pt, jbarbosa@fe.up.pt*

The task scheduling problem is broadly classified into two major categories, namely Static Scheduling and Dynamic Scheduling. In the Static category, all information about tasks such as execution and communication costs for each task and the relationship with other tasks is known beforehand; in the dynamic category, such information is not available and decisions are made at runtime. Moreover, Static scheduling is an example of compile-time scheduling, whereas Dynamic scheduling is representative of run-time scheduling. Static scheduling algorithms are universally classified into two major groups, namely Heuristic-based and Guided Random Search-based algorithms. Heuristic-based algorithms allow approximate solutions, often good solutions, with polynomial time complexity. Guided Random Search-based algorithms also give approximate solutions, but the solution quality can be improved by including more iterations, which therefore makes them more expensive than the Heuristic-based approach [29]. The Heuristic-based group is composed of three subcategories: list, clustering and duplication scheduling. Clustering heuristics are mainly proposed for homogeneous systems to form clusters of tasks that are then assigned to processors. For heterogeneous systems, CHP algorithms [8] and Triplet [13] were proposed, but they have limitations in higher-heterogeneity systems. The duplication heuristics produce the shortest *makespans*, but they have two disadvantages: a higher time complexity, i.e., cubic, in relation to the number of tasks, and the duplication of the execution of tasks, which results in more processor power used. This is an important characteristic not only because of the associated energy cost but also because, in a shared resource, fewer processors are available to run other

concurrent applications. List scheduling heuristics, on the other hand, produce the most efficient schedules, without compromising the *makespan* and with a complexity that is generally quadratic in relation to the number of tasks. HEFT has a complexity of $O(v^2.p)$, where v is the number of tasks and p is the number of processors.

In this paper, we present a new list scheduling algorithm for a bounded number of fully connected heterogeneous processors called Predict Earliest Finish Time (PEFT) that outperforms state-of-the-art algorithms such as HEFT in terms of *makespan* and Efficiency. The time complexity is $O(v^2.p)$, as in HEFT. To our knowledge, this is the first algorithm to outperform HEFT while having the same time complexity. We also introduce one innovation, a look ahead feature, without increasing the time complexity. Other algorithms such as LDCP [15] and Lookahead [7] have this feature but with a cubic and quartic time complexity, respectively. We present results for randomly generated DAGs [28] and DAGs from well-known applications used in related papers, such as Gaussian Elimination, Fast Fourier Transform, Montage and Epigenomic Workflows.

This paper is organized as follows: in Section 2, we introduce the task scheduling problem; in Section 3, we present related work in scheduling DAGs on heterogeneous systems, and we present in detail the scheduling algorithms that are used for the comparison with PEFT, which is the list scheduling heuristic proposed here; in Section 4, we present PEFT; in Section 5 we present the results and, finally, we present the conclusions in Section 6.

2 Scheduling problem formulation

The problem addressed in this paper is the static scheduling of a single application in a heterogeneous system with a set P of processors. As mentioned above, task scheduling can be divided into Static and Dynamic approaches. Dynamic scheduling is adequate for situations where the system and task parameters are not known at compile time, which requires decisions to be made at runtime but with additional overhead. A sample environment is a system where users submit jobs, at any time, to a shared computing resource [1]. A dynamic algorithm is required because the workload is only known at runtime, as is the status of each processor when new tasks arrive. Consequently, a dynamic algorithm does not have all work requirements available during scheduling and cannot optimize based on the entire workload. By contrast, a static approach can maximize a schedule by considering all tasks independently of execution order or time because the schedule is generated before execution begins and introduces no overhead at runtime. In this paper, we present an algorithm that minimizes the execution time of a single job on a set of P processors. We consider that P processors are available for the job and that they are not shared during the job execution. Therefore, with the system and job parameters known at compile time, a static scheduling approach has no overhead at runtime and is more appropriate.

An application can be represented by a *Directed Acyclic Graph* (DAG), $G = (V, E)$, as shown in Figure 1, where V is the set of v nodes and each node $v_i \in V$ represents an application task, which includes instructions that must be executed on the same machine. E is the set of e communication edges between tasks; each $e(i, j) \in E$ represents the task-dependency constraint such that task n_i should complete its execution before task n_j can be started. The DAG is complemented by a matrix W that is a $v \times p$ computation cost matrix, where v is the number of tasks and p is the number of processors in the system. $w_{i,j}$ gives the estimated time to execute task v_i on machine p_j. The mean execution time of task v_i is calculated by equation 1.

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

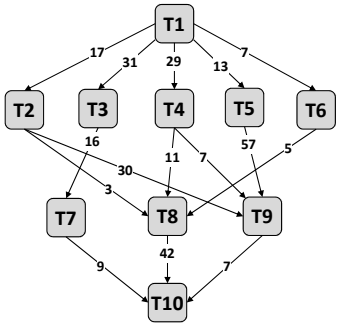
The average execution time $\overline{w_i}$ is commonly used to compute a priority rank for the tasks. The algorithm proposed in this paper uses the $w_{i,j}$ rather than $\overline{w_i}$ as explained in section 4.

Each edge $e(i, j) \in E$ is associated with a non-negative weight $c_{i,j}$ representing the communication cost between the tasks v_i and v_j. Because this value can be computed only after defining where tasks v_i and v_j will be executed, it is common to compute the average communication costs to label the edges [29]. The average communication cost $\overline{c_{i,j}}$ of an edge $e(i, j)$ is calculated by equation 2.

$$\overline{c_{i,j}} = \overline{L} + \frac{data_{i,j}}{\overline{B}} \tag{2}$$

where $\overline{L}$ is the average latency time of all processors and $\overline{B}$ is the average bandwidth of all links connecting the set of P processors. $data_{i,j}$ is the amount of data elements that task v_i needs to send to task v_j. Note that when tasks v_i and v_j are assigned to the same processor, the real communication cost is considered to be zero because it is negligible compared with interprocessor communication costs.

Additionally, in our model, we consider processors that are connected in a fully connected topology. The execution of tasks and communications with other processors can be achieved for each processor simultaneously and without contention. Additionally, the execution of any task is considered nonpreemptive. These model simplifications are common in this scheduling problem [15], [20], [29], and we consider them to permit a fair comparison with state-of-the-art algorithms and because these simplifications correspond to real systems. Our target system is a single



Task	P1	P2	P3
T1	22	21	36
T2	22	18	18
T3	32	27	43
T4	7	10	4
T5	29	27	35
T6	26	17	24
T7	14	25	30
T8	29	23	36
T9	15	21	8
T10	13	16	33

Fig. 1: Application DAG and computation time matrix of the tasks in each processor for a three processor machine

site infrastructure that can be as simple as a set of devices (e.g., CPUs and GPUs) connected by a switched network that guarantees parallel communications between different pairs of devices. The machine is heterogeneous because CPUs can be from different generations and also because other very different devices such as GPUs can be included. Another common machine is the one that results from selecting processors from several clusters from the same site. Although a cluster is homogeneous, the set of processors selected to execute a given DAG forms a heterogeneous machine. Because the clusters are connected by high-speed networks, with redundant links, the simplification is still reasonable. The processor latency can differ in a heterogeneous machine, but such differences are negligible. For low communication-to-computation ratios (CCRs), the communications are negligible; for higher CCRs, the predominant factor is the network bandwidth, and we consider that the bandwidth is the same throughout the entire network.

Next, we present some of the common attributes used in task scheduling, which we will refer to in the following sections.

Definition (1) $pred(n_i)$: denotes the set of immediate predecessors of task n_i in a given DAG. A task with no predecessors is called an $entry$ task, n_{entry}. If a DAG has multiple entry nodes, a dummy entry node with zero weight and zero communication edges can be added to the graph.

Definition (2) $succ(n_i)$: denotes the set of immediate successors of task n_i. A task with no successors is called an $exit$ task, n_{exit}. Similar to the entry node, if a DAG has multiple exit nodes, a dummy exit node with zero weight and zero communication edges from current multiple exit nodes to this dummy node can be added to the graph.

Definition (3) *makespan* or $schedule\ length$: denotes the finish time of the last task in the scheduled DAG and is defined by

$$makespan = max\{AFT(n_{exit})\} \tag{3}$$

where $AFT(n_{exit})$ denotes the *Actual Finish Time* of the exit node. In the case where there is more than one exit node and no redundant node is added, the *makespan* is the maximum actual finish time of all exit tasks.

Definition (4) $level(n_i)$: the level of task n_i is an integer value representing the maximum number of edges of the paths from the entry node to n_i. For the entry node, the level is $level(n_{entry}) = 1$, and for other tasks, it is given by

$$level(n_i) = \max_{q \in pred(n_i)} \{level(q)\} + 1 \tag{4}$$

Definition (5) $Critical\ Path(CP)$: the CP of a DAG is the longest path from the $entry$ node to the $exit$ node in the graph. The lower bound of a schedule length is the minimum critical path length (CP_{MIN}), which is computed by considering the minimum computational costs of each node in the critical path.

Definition (6) $EST(n_i, p_j)$: denotes the *Earliest Start Time* of a node n_i on a processor p_j and is defined as

$$EST(n_i, p_j) = \max\left\{T_{Available}(p_j), \max_{n_m \in pred(n_i)}\left\{AFT(n_m) + c_{m,i}\right\}\right\} \tag{5}$$

where $T_{Available}(p_j)$ is the earliest time at which processor p_j is ready. The inner max block in the EST equation is the time at which all data needed by n_i arrive at the processor p_j. The communication cost $c_{m,i}$ is zero if the predecessor node n_m is assigned to processor p_j. For the entry task, $EST(n_{entry}, p_j) = 0$.

Definition (7) $EFT(n_i, p_j)$: denotes the *Earliest Finish Time* of a node n_i on a processor p_j and is defined as

$$EFT(n_i, p_j) = EST(n_i, p_j) + w_{i,j} \tag{6}$$

which is the *Earliest Start Time* of a node n_i on a processor p_j plus the computational cost of n_i on a processor p_j.

The *objective* of the scheduling problem is to determine an assignment of tasks of a given DAG to processors such that the *Schedule Length* is minimized. After all nodes in the DAG are scheduled, the schedule length will be the *Actual Finish Time* of the exit task, as expressed by equation 3.

3 Related Work

In this section, we present a brief survey of task scheduling algorithms, specifically list-based heuristics. We present their time complexity and their comparative performance.

Over the past few years, research on static DAG scheduling has focused on finding suboptimal solutions to obtain a good solution in an acceptably short time. List scheduling heuristics usually generate high-quality schedules at a reasonable cost. In comparison with clustering algorithms, they have lower time

complexity, and in comparison with task duplication strategies, their solutions represent a lower processor workload.

3.1 List-based algorithms

Many list scheduling algorithms have been developed by researchers. This type of scheduling algorithm has two phases: the *prioritizing* phase for giving a priority to each task and a *processor selection* phase for selecting a suitable processor that minimizes the heuristic cost function. If two or more tasks have equal priority, then the tie is resolved by selecting a task randomly. The last phase is repeated until all tasks are scheduled to suitable processors. Table 1 presents some list scheduling algorithms including some of the most cited, along with their time complexity.

Algorithm	Reference	Complexity
MH	[17]	$O(v^2.p)$
DLS	[27]	$O(v^3.p)$
LMT	[23]	$O(v^2.p^2)$
BIL	[24]	$O(v^2.p.\log p)$
FLB	[25]	$O(v.(\log p + \log v) + v^2)$
CPOP	[30], [29]	$O(v^2.p)$
HEFT	[30], [29]	$O(v^2.p)$
HCPT	[19]	$O(v^2.p)$
HPS	[22]	$O(v^2(p.\log v))$
PETS	[21]	$O(v^2(p.\log v))$
LDCP	[15]	$O(v^3.p)$
Lookahead	[7]	$O(v^4.p^3)$

TABLE 1: List-based scheduling algorithms for heterogeneous systems

The MH and DLS algorithms consider a link contentions model, but their versions for a fully connected network were used for comparison purposes by other authors and are therefore described here. The schedules generated by the Mapping Heuristic (MH) algorithm [17] are generally longer than recently developed heuristics because MH only considers a processor ready when it finishes the last task assigned to it. Therefore, MH ignores the possibility that a processor that is busy when a task is being scheduled can complete the task in a shorter time, which thus results in poorer scheduling. The time complexity of MH without contention is $O(v^2.p)$ and $O(v^2.p^3)$ otherwise. Dynamic Level Scheduling (DLS) [27] is one of the first algorithms that computed an estimate of the availability of each processor and thus allowed a task to be scheduled to a currently busy processor. Consequently, DLS yields better results than MH. The DLS authors proposed a version for a heterogeneous machine, but the processor selection was based on the *Earliest Start Time* (EST) as the homogeneous version, which is one of the drawbacks of the algorithm because the EST does not guarantee the minimum completion time for a task. Additionally, DLS does not try to fill scheduling holes in a processor schedule (idle time slots that are between two tasks already scheduled on the same processor), in contrast to other more recent algorithms. The time complexity of DLS for a fully connected network is $O(v^3.p)$, where the routing complexity is equal to 1. The Levelized min-Time Algorithm (LMT) [23] is a very simple heuristic that assigns priorities to tasks based on their precedence constraints, which are called levels. The time complexity is squared in relation to the number of processors and tasks. The schedules produced are significantly worse than those produced by other more recently developed heuristics, such as CPOP [30], [29]. The Best Imaginary Level (BIL) [24] defines a static level for DAG nodes, called BIL, that incorporates the effect of interprocessor communication overhead and processor heterogeneity. The BIL heuristic provides an optimal schedule for linear DAGs. Fast Load Balancing (FLB) [25] was proposed with the aim of reducing the time complexity relative to that of HEFT [30], [29]. FLB generates schedules comparable to those of HEFT, but it generates poor schedules for irregular task graphs and for higher processor heterogeneities. The Critical Path On a Processor (CPOP) [30], [29] achieves better schedules than LMT and MH as well as schedules that are comparable to those of DLS, with a lower time complexity. The main feature of CPOP is the assignment of all the tasks that belong to the critical path to a single processor. Heterogenous Critical Parent Trees (HCPT) [19] yield better scheduling results than CPOP, FLB and DLS. The Heterogeneous Earliest-Finish-Time (HEFT) [30], [29] is one of the best list scheduling heuristics, as it has quadratic time complexity. In [11], the authors compared 20 heuristics and concluded that HEFT produces the shortest schedule lengths for random graphs.

HPS[22] and PETS[21] are other list scheduling heuristics reported to achieve better results than HEFT. HPS, PETS, HEFT, HCPT and Lookahead are explained in more detail in section 3.2, and they are used in this paper for comparison with our proposed list scheduling algorithm.

For the Longest Dynamic Critical Path (LDCP) [15], the authors reported better scheduling results than HEFT, although these results were not significant when considering the associated increase in complexity. For random graphs, the improvements in the schedule length ratio over HEFT were less than 3.1%. The algorithm builds for each processor a DAG, called a DAGP, that consists of the initial DAG with the computation costs of the processors. The complexity is higher (cubic) because the algorithm needs to update all DAGPs after scheduling a task to a processor.

Another recent algorithm that reported an average improvement in *makespan* over HEFT is the Lookahead approach [7]. This algorithm has quartic complexity for the one step Lookahead. We consider the Lookahead for comparison with our algorithm be-

cause it has achieved the best results reported thus far in the literature. Because it has a higher complexity, it serves as a reference and an upper bound for our algorithm.

3.2 Selected list scheduling heuristics

Here, we describe the list scheduling heuristics for scheduling tasks on a bounded number of heterogeneous processors selected for comparison with our proposed algorithm, namely HEFT, HCPT, HPS, PETS and Lookahead.

3.2.1 Heterogeneous Earliest Finish Time (HEFT)

The HEFT algorithm [29] is highly competitive in that it generates a schedule length comparable to the schedule lengths of other scheduling algorithms with a lower time complexity. The HEFT algorithm has two phases: a *task prioritizing* and a *processor selection* phase. In the first phase task, priorities are defined as $rank_u$. $rank_u$ represents the length of the longest path from task n_i to the exit node, including the computational cost of n_i, and is given by $rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)}\{\overline{c_{i,j}} + rank_u(n_j)\}$. For the exit task, $rank_u(n_{exit}) = \overline{w_{exit}}$. The task list is ordered by decreasing value of $rank_u$. In the *processor selection phase*, the task on top of the task list is assigned to the processor p_j that allows for the EFT (Earliest Finish Time) of task n_i. However, the HEFT algorithm uses an insertion policy that tries to insert a task in at the earliest idle time between two already scheduled tasks on a processor, if the slot is large enough to accommodate the task.

3.2.2 Heterogeneous Critical Parent Trees (HCPT)

The HCPT algorithm [19] uses a new mechanism to construct the scheduling list L, rather than assigning priorities to the application tasks. HCPT divides the task graph into a set of unlisted-parent trees. The root of each unlisted-parent tree is a critical path node (CN). A CN is defined as the node that has zero difference between its *AEST* and *ALST*. The *AEST* is the *Average Earliest Start Time* of node n_i and is equivalent to $rank_d$, as indicated by $AEST(n_i) = \max_{n_j \in pred(n_i)}\{AEST(n_j) + \overline{w_j} + \overline{c_{j,i}}\}$; $AEST(n_{entry}) = 0$.

The *Average Latest Start Time* ($ALST$) of node n_i can be computed recursively by traversing the DAG upward starting from the exit node and is given by $ALST(n_i) = \min_{n_j \in succ(n_i)}\left\{ALST(n_j) - \overline{c_{i,j}}\right\} - \overline{w_i}$; $ALST(n_{exit}) = AEST(n_{exit})$.

The algorithm also has two phases, namely *listing tasks* and *processor assignment*. In the first phase, the algorithm starts with an empty queue L and an auxiliary stack S that contains the CNs pushed in decreasing order of their ALSTs, i.e., the entry node is on top of S. Consequently, $top(S)$ is examined. If $top(S)$ has an unlisted parent (i.e., has a parent not in L), then this parent is pushed on the stack S. Otherwise, $top(S)$ is removed and enqueued into L. In the processor assignment phase, the algorithm tries to assign each task $n_i \in L$ to a processor p_j that allows the task to be completed as early as possible.

3.2.3 High Performance Task Scheduling (HPS)

The HPS [22] algorithm has three phases, namely a *level sorting*, *task prioritization* and *processor selection* phase. In the level sorting phase, the given DAG is traversed in a top-down fashion to sort tasks at each level to group the tasks that are independent of each other. As a result, tasks in the same level can be executed in parallel. In the task prioritization phase, priority is computed and assigned to each task using the attributes *Down Link Cost* (DLC), *Up Link Cost* (ULC) and *Link Cost* (LC) of the task. The DLC of a task is the maximum communication cost among all the immediate predecessors of the task. The DLC for all tasks at level 0 is 0. The ULC of a task is the maximum communication cost among all the immediate successors of the task. The ULC for an exit task is 0. The LC of a task is the sum of DLC, ULC and maximum LC for all its immediate predecessor tasks.

At each level, based on the LC values, the task with the highest LC value receives the highest priority, followed by the task with the next highest LC value and so on in the same level. In the processor selection phase, the processor that gives the minimum *EFT* for a task is selected to execute that task. HPS has an insertion-based policy, which considers the insertion of a task in the earliest idle time slot between two already-scheduled tasks on a processor.

3.2.4 Performance Effective Task Scheduling (PETS)

The PETS algorithm [21] has the same three phases as HPS. In the level sorting phase, similar to HPS, tasks are categorized in levels such that in each level, the tasks are independent. In the task prioritization phase, priority is computed and assigned to each task using the attributes *Average Computation Cost* (ACC), *Data Transfer Cost* (DTC) and the *Rank of Predecessor Task* (RPT). The ACC of a task is the average computation cost for all p processors, which we referred to before as $\overline{w_i}$. The DTC of a task n_i is the communication cost incurred when transferring the data from task n_i to all its immediate successor tasks; for an exit node, $DTC(n_{exit}) = 0$. The RPT of a task n_i is the highest rank of all its immediate predecessor tasks; for an entry node, $RPT(n_{entry}) = 0$. The rank is computed for each task n_i based on the tasks ACC, DTC and RPT values and is given by $rank(n_i) = round\{ACC(n_i) + DTC(n_i) + RPT(n_i)\}$.

At each level, the task with the highest rank value receives the highest priority, followed by the task with next highest rank value and so on. A tie is broken by selecting the task with the lower ACC value. As

in some of the other task scheduling algorithms, in the processor selection phase, this algorithm selects the processor that gives the minimum *EFT* value for executing the task. It also uses an insertion-based policy for scheduling a task in an idle slot between two previously scheduled tasks on a given processor.

3.2.5 Lookahead Algorithm

The Lookahead scheduling algorithm [7] is based on the HEFT algorithm, whose main feature is its processor selection policy. To select a processor for the current task t, it iterates over all available processors and computes the EFT for the child tasks on all processors. The processor selected for task t is the one that minimizes the maximum EFT from all children of t on all resources where t is tried. This procedure can be repeated for each child of t by increasing the number of levels analyzed. In HEFT, the complexity of v tasks is $O(e.p)$, where EFT is computed v times. In the worst case, by replacing e by v^2, we obtain $O(v^2.p)$. The Lookahead algorithm has the same structure as HEFT but computes EFT for each child of the current task. The number of EFT calls (graph vertices) is equal to $v + p.e$ for a single level of forecasting. By replacing this number of vertices in $O(v^2.p)$, in the worst case the total time complexity of Lookahead is $O(v^4.p^3)$. The authors reported that additional levels of forecasting do not result in significant improvements in the makespan. Here, we only consider the one-level Lookahead.

4 The Proposed Algorithm PEFT

In this section, we introduce a new list-based scheduling algorithm for a bounded number of heterogeneous processors, called PEFT. The algorithm has two major phases: a *task prioritizing* phase for computing task priorities, and a *processor selection* phase for selecting the best processor for executing the current task.

In our previous work [5], we evaluated the performance of list-based scheduling algorithms. We compared their results with the solutions achieved by three meta-heuristic algorithms, namely Tabu Search, Simulated Annealing and Ant Colony System. The meta-heuristic algorithms, which feature a higher processing time, always achieved better solutions than the list scheduling heuristics with quadratic complexity. We then compared the best solutions for both types, step by step. We observed that the best meta-heuristic schedules could not be achieved if we followed the common strategy of selecting processors based only on current task execution time, because the best schedules consider not only the immediate gain in processing time but also the gain in a sequence of tasks. Most list-based scheduling heuristics with quadratic time complexity assign a task to a processor by evaluating only the current task. This methodology, although inexpensive, does not evaluate what is ahead of the current task, which leads to poor decisions in some cases. Algorithms that analyze the impact on children nodes, such as Lookahead [7], exist, but they increase the time complexity to the 4th order.

The most powerful feature of the Lookahead algorithm, as the best algorithm with the lowest makespan, is its ability to forecast the impact of an assignment for all children of the current task. This feature permits better decisions to made in selecting processors, but it increases the complexity significantly. Therefore, the novelty of the proposed algorithm is its ability to forecast by computing an *Optimistic Cost Table* while maintaining quadratic time complexity, as explained in the following section.

4.1 Optimistic cost table (OCT)

Our algorithm is based on the computation of a cost table on which task priority and processor selection are based. The OCT is a matrix in which the rows indicate the number of tasks and the columns indicate the number of processors, where each element $OCT(t_i, p_k)$ indicates the maximum of the shortest paths of t_i children's tasks to the exit node considering that processor p_k is selected for task t_i. The OCT value of task t_i on processor p_k is recursively defined by Equation 7 by traversing the DAG from the exit task to the entry task.

$$OCT(t_i, p_k) = \max_{t_j \in succ(t_i)} \left[\min_{p_w \in P} \left\{ OCT(t_j, p_w) + w(t_j, p_w) + \overline{c_{i,j}} \right\} \right], \quad \overline{c_{i,j}} = 0 \; if \; p_w = p_k. \tag{7}$$

where $\overline{c_{i,j}}$ is the average communication cost, which is zero if t_j is being evaluated for processor p_k, and $w(t_j, p_w)$ is the execution time of task t_j on processor p_w. As explained before, we use the average communication cost and the execution cost for each processor. $OCT(t_i, p_k)$ represents the maximum optimistic processing time of the children of task t_i because it considers that children tasks are executed in the processor that minimizes processing time (communications and execution) independently of processor availability, as the OCT is computed before scheduling begins. Because it is defined recursively and the children already have the optimistic cost to the exit node, only the first level of children is considered. For the exit task, the $OCT(n_{exit}, p_k) = 0$ for all processors $p_k \in P$.

4.2 Task prioritizing phase

To define task priority, we compute the average OCT for each task that is defined by equation 8.

$$rank_{oct}(t_i) = \frac{\sum_{k=1}^{P} OCT(t_i, p_k)}{P} \quad (8)$$

Table 2 shows the values of OCT for the DAG sample of Figure 1. When the new rank $rank_{oct}$ is compared with $rank_u$, the former shows slight differences in the order of the tasks based on these two priority strategies. For instance, $T5$ has a lower $rank_{oct}$ value than $T4$, where $T4$ is selected first for scheduling. With $rank_u$, the opposite is true. The main feature of our algorithm is the cost table that reflects for each task and processor the cost to execute descendant tasks until the exit node. This information permits an informed decision to be made in assigning a processor to a task. Task ranking is a less relevant issue because few tasks in each step are ordered by priority and the influence on performance is less relevant. By comparing $rank_u$ and $rank_{oct}$, we can see that $rank_u$ uses the average computing cost for each task and also accumulates the maximum descendant costs of descendant tasks to the exit node. In contrast, $rank_{oct}$ is an average over a set of values that were computed with the cost of each task on each processor. Therefore, the ranks are computed using a similar procedure, and significant differences in performance are not expected when using either system.

For the tests with random graphs reported in the results section, when using $rank_u$ the performance is on average better in approximately 0.5%. The OCT exerts a greater influence in the processor selection phase, and using $rank_u$ would require additional computations without providing a significant advantage.

Task	$P1$	$P2$	$P3$	$rank_{oct}$	$rank_u$
T1	64	68	86	72.7	169
T2	42	39	42	41	114.3
T3	27	41	43	37	102.7
T4	42	39	50	43.7	110
T5	28	37	28	31	129.7
T6	42	39	44	41.7	119.3
T7	13	16	22	17	52.7
T8	13	16	33	20.7	92
T9	13	16	20	16.3	42.3
T10	0	0	0	0	20.7

TABLE 2: Optimistic Cost Table for the DAG of Figure 1; $rank_{oct}$ and $rank_u$ are also shown for comparison purposes

4.3 Processor selection phase

To select a processor for a task, we compute the Optimistic EFT (O_{EFT}), which sums to EFT the computation time of the longest path to the exit node. In this way, we are looking forward (forecasting) in the processor selection; perhaps we are not selecting the processor that achieves the earliest finish time for the current task, but we expect to achieve a shorter finish time for the tasks in the next steps. The aim is to guarantee that the tasks ahead will finish earlier, which is the purpose of the OCT table. O_{EFT} is defined by equation 9.

$$O_{EFT}(t_i, p_j) = EFT(t_i, p_j) + OCT(t_i, p_j) \quad (9)$$

4.4 Detailed description of PEFT algorithm

In this section, we present the algorithm PEFT, supported by an example, to detail the description of each step. The proposed PEFT algorithm is formalized in Algorithm 1.

Algorithm 1 The PEFT algorithm

1: Compute OCT table and $rank_{oct}$ for all tasks
2: Create Empty list *ready-list* and put n_{entry} as initial task
3: **while** *ready-list* is NOT Empty **do**
4: $n_i \leftarrow$ the task with highest $rank_{oct}$ from *ready-list*
5: **for all** processor p_j in the processor-set P **do**
6: Compute $EFT(n_i, p_j)$ value using insertion-based scheduling policy
7: $O_{EFT}(n_i, p_j) = EFT(n_i, p_j) + OCT(n_i, p_j)$
8: **end for**
9: Assign task n_i to the processor p_j that minimize O_{EFT} of task n_i
10: Update *ready-list*
11: **end while**

The algorithm starts by computing the OCT table and $rank_{oct}$ at line 1. It then creates an empty $ready - list$ and places the entry task on top of the list. In the while loop, from line 4 to 10, in each iteration, the algorithm will schedule the task with a higher value of $rank_{oct}$. After selecting the task for scheduling, the PEFT algorithm calculates the O_{EFT} values for the task on all processors. In the processor selection phase, the aim is to guarantee that the tasks ahead will finish earlier, but rather than analyzing all tasks until the end, to reduce complexity we use the OCT table, which incorporates that information. In line 9, the processor p_j that achieves the minimum $O_{EFT}(n_i, p_j)$ is selected to execute task n_i.

Table 3 shows an example that demonstrates the PEFT for the DAG of Figure 1.

Figure 2 shows the scheduling results for the sample DAG with the algorithms PEFT, Lookahead, HEFT, HCPT, HPS and PETS. By comparing the schedules of PEFT and HEFT, we can see that T1 is assigned to P1 although it does not guarantee the earliest finish time for T1, but P1 minimizes the expected EFT of all DAGs. This is only one example used to illustrate the algorithm, but as shown in the results section, PEFT produces, on average, better schedules than the state-of-the-art algorithms.

Step	Ready List	Task selected	EFT			O_{EFT}			CPU Selected
			P1	P2	P3	P1	P2	P3	
1	T1	T1	22	**21**	36	**86**	89	122	P1
2	T4,T6,T2,T3,T5	T4	**29**	61	55	**71**	100	105	P1
3	T6,T2,T3,T5	T6	55	**46**	53	97	**85**	97	P2
4	T2,T3,T5	T2	**51**	64	57	**93**	103	99	P1
5	T3,T5,T8	T3	83	**80**	96	**110**	121	139	P1
6	T5,T8,T7	T5	112	73	**70**	140	110	**98**	P3
7	T8,T7,T9	T8	112	**77**	106	125	**93**	139	P2
8	T7,T9	T7	**97**	124	129	**110**	140	151	P1
9	T9	T9	142	148	**89**	155	164	**109**	P3
10	T10	T10	132	**122**	152	132	**122**	152	P2

TABLE 3: Schedule produced by the PEFT algorithm in each iteration

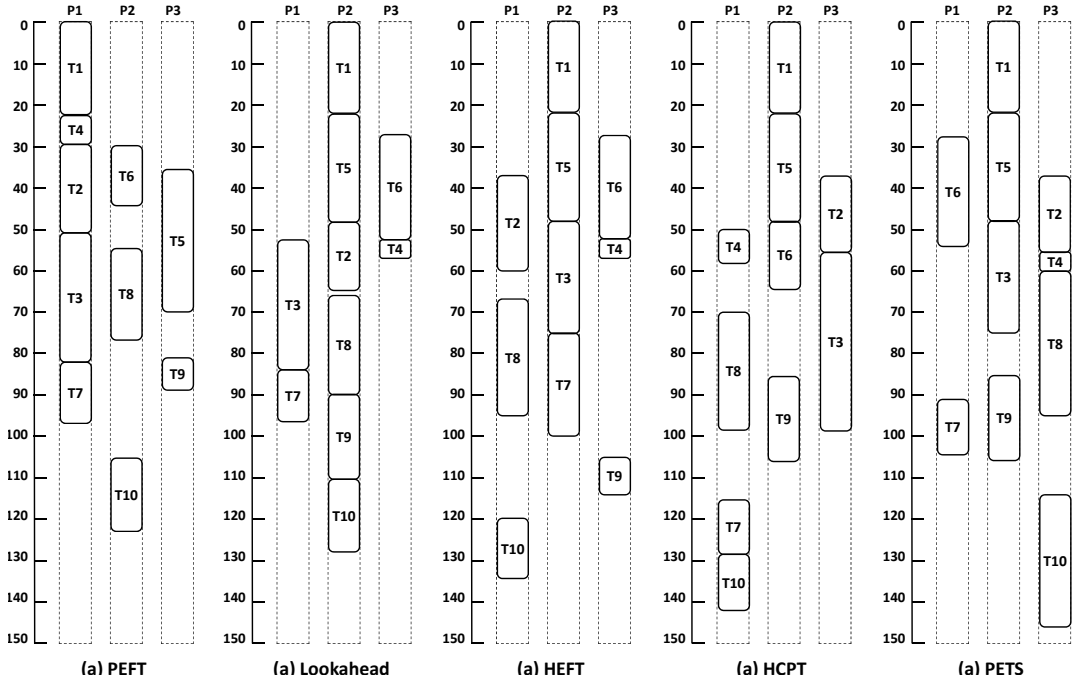


Fig. 2: Schedules of the sample task graph in Figure 1 with **(a)** PEFT (*makespan=122*), **(b)** Lookahead (*makespan=127*), **(c)** HEFT (*makespan=133*), **(d)** HCPT (*makespan=142*), **(e)** PETS (*makespan=147*)

In terms of time complexity, PEFT requires the computation of an OCT table that is $O(p(e+v))$, and to assign the tasks to processors the time complexity is of the order $O(v^2.p)$. The total time is $O(p(e+v)+v^2.p)$. For dense DAGs, e becomes v^2, where the total algorithm complexity is of the order $O(v^2.p)$. That is, the time complexity of PEFT is of the same order as the HEFT algorithm.

5 Experimental Results and Discussion

This section compares the performance of the PEFT algorithm with that of the algorithms presented above. For this purpose, we consider two sets of graphs as the workload: randomly generated application graphs and graphs that represent some real-world applications. We first present the comparison metrics used for the performance evaluation.

5.1 Comparison Metrics

The comparison metrics are Scheduling Length Ratio, Efficiency, pair-wise comparison of the number of occurrences of better solutions and Slack.

Scheduling Length Ratio (SLR)

The metric most commonly used to evaluate a schedule for a single DAG is the *makespan*, as defined by equation 3. Here, we want to use a metric that compares DAGs with very different topologies; the metric most commonly used to do so is the Normalized Schedule Length (NSL) [15], which is also called the

Scheduling Length Ratio (SLR) [29]. For a given DAG, both represent the *makespan* normalized to the lower bound. SLR is defined by equation 10.

$$SLR = \frac{makespan(solution)}{\sum_{n_i \in CP_{MIN}} \min_{p_j \in P} (w_{(i,j)})} \quad (10)$$

The denominator in SLR is the minimum computation cost of the critical path tasks (CP_{MIN}). There is no *makespan* less than the denominator of the SLR equation. Therefore, the algorithm with the lowest SLR is the best algorithm.

Efficiency

In the general case, Efficiency is defined as the Speedup divided by the number of processors used in each run, and Speedup is defined as the ratio of the sequential execution time to the parallel execution time (i.e., the *makespan*). The sequential execution time is computed by assigning all tasks to a single processor that minimizes the total computation cost of the task graph, as shown by equation 11.

$$Speedup = \frac{\min\limits_{p_j \in P} \left[\sum\limits_{n_i \in V} w_{(i,j)} \right]}{makespan(solution)} \quad (11)$$

Number of occurrences of better solutions

This comparison is presented as a pair-wise table, where the percentage of better, equal and worse solutions produced by PEFT is compared to that of the other algorithms.

Slack

The *slack* metric [9], [26] is a measure of the robustness of the schedules produced by an algorithm to uncertainty in the tasks processing time, and it represents the capacity of the schedule to absorb delays in task execution. The *slack* is defined for a given schedule and a given set of processors. The *slack* of a task represents the time window within which the task can be delayed without extending the makespan. *Slack* and makespan are two conflicting metrics; lower makespans produce small *slack*. For deterministic schedules, the *slack* is defined by Equation12.

$$Slack = \Big[\sum_{t_i \in V} M - b_{level}(t_i) - t_{level}(t_i) \Big] / n \quad (12)$$

where M is the makespan of the DAG, n is the number of tasks, b_{level} is the length of the longest path to the exit node and t_{level} is the length of the longest path from the entry node. These values are referred to a given schedule, and therefore the processing time used for each task is the processing time on the processor that it was assigned. The aim of using this metric is to evaluate whether the proposed algorithm has an equivalent *slack* to HEFT, which is the reference quadratic algorithm.

5.2 Random Graph Generator

To evaluate the relative performance of the heuristics, we first considered randomly generated application graphs. For this purpose, we used a synthetic DAG generation program available at [28] with an adaptation to the fat parameter, as explained next. Five parameters define the DAG shape:

- ***n***: number of computation nodes in the DAG (i.e., application tasks);
- ***fat***: this parameter affects the height and the width of the DAG. The width in each level is defined by a uniform distribution with a mean equal to $fat.\sqrt{n}$. The height, or the number of levels, is created until n tasks are defined in the DAG. The width of the DAG is the maximum number of tasks that can be executed concurrently. A small value will lead to a thin DAG (e.g., chain) with low task parallelism, whereas a large value induces a fat DAG (e.g., fork-join) with a high degree of parallelism;
- ***density***: determines the number of edges between two levels of the DAG, with a low value leading to few edges and a large value leading to many edges;
- ***regularity***: the regularity determines the uniformity of the number of tasks in each level. A low value indicates that levels contain dissimilar numbers of tasks, whereas a high value indicates that all levels contain similar numbers of tasks;
- ***jump***: indicates that an edge can go from level l to level $l + jump$. A jump of 1 is an ordinary connection between two consecutive levels.

In the present study, we used this synthetic DAG generator to create the DAG structure, which includes the specific number of nodes and their dependencies. To obtain computation and communication costs, the following parameters are used:

- ***CCR*** (Communication to Computation Ratio): ratio of the sum of the edge weights to the sum of the node weights in a DAG;
- β (Range percentage of computation costs on processors): the *heterogeneity factor* for processor speeds. A high β value implies higher heterogeneity and different computation costs among processors, and a low value implies that the computation costs for a given task are nearly equal among processors [29]. The average computation cost of a task n_i in a given graph $\overline{w_i}$ is selected randomly from a uniform distribution with range $\left[0, 2 \times \overline{w_{DAG}}\right]$, where $\overline{w_{DAG}}$ is the average computation cost of a given graph that is obtained randomly. The computation cost of each task n_i on each processor p_j is randomly set from the range of equation 13.

$$\overline{w_i} \times \left(1 - \frac{\beta}{2}\right) \leq w_{i,j} \leq \overline{w_i} \times \left(1 + \frac{\beta}{2}\right) \quad (13)$$

In our experiment, for random DAG generation, we considered the following parameters:

- $n = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200\ 300, 400, 500]$
- $CCR = [0.1, 0.5, 0.8, 1, 2, 5, 10]$
- $\beta = [0.1, 0.2, 0.5, 1, 2]$
- $jump = [1, 2, 4]$
- $regularity = [0.2, 0.8]$
- $fat = [0.1, 0.4, 0.8]$
- $density = [0.2, 0.8]$
- $Processors = [4, 8, 16, 32]$

These combinations produce 70,560 different DAGs. For each DAG, 10 different random graphs were generated with the same structure but with different edge and node weights. Thus, 705,600 random DAGs were used in the study.

Figure 3a shows the average SLR and Figure 3b shows the average *slack* for all algorithms as a function of the DAG size. For DAGs featuring up to 100 tasks, Lookahead and PEFT present similar results. For larger DAGs, PEFT is the best algorithm, as it outperforms the Lookahead algorithm. All quadratic algorithms maintain a certain level of performance, in contrast to the Lookahead algorithm, which after 100 tasks suffers a substantial decrease in performance. The Lookahead algorithm bases its decisions on the analysis of the children nodes for the current task, expecting that those nodes will be scheduled shortly. However, if there are too many concurrent tasks to schedule, as observed for DAGs with more than 100 tasks, the processor load is substantially changed by the concurrent tasks to be scheduled after the current task. This finding implies that when the children tasks are scheduled, the conditions are different and the optimization decision made by the parent task is not valid, which results in poorer performance. Compared with HEFT, our PEFT algorithm improves by 10% for 10 task DAGs. This improvement gradually decreases to 6.2% for 100 task DAGs and to 4% for 500 task DAGs. Despite this significant improvement, we can observe that PEFT maintains the same level of *Slack* as HEFT. Thus, the schedules produced, although shorter, have the same robustness to uncertainty as those produced by HEFT.

To illustrate the results statistically, boxplots are presented, where the minimum, 25%, mean and 75% values of the algorithm results are represented. The maximum is not shown because there is a broad distribution in the results; therefore, we only show values up to the Upper Inner Fence. We also show the average values, which are indicated by an individual line in each boxplot.

The SLRs obtained for the PEFT, Lookahead, HEFT, HCPT, HPS and PETS algorithms as a function of CCR and heterogeneity are shown in Figure 4a and 4b, respectively. We can see that PEFT has the lowest average SLR and a smaller dispersion in the distribution of the results. The second best algorithm in terms of SLR is Lookahead. In terms of Efficiency, Figure 4c, Lookahead is the best algorithm, with a performance very similar to that of PEFT. This is an important finding because with the proposed algorithm, we improved the SLR and also achieved high values of Efficiency that are only comparable with those of a higher-complexity algorithm. PEFT was the best quadratic algorithm in our simulation.

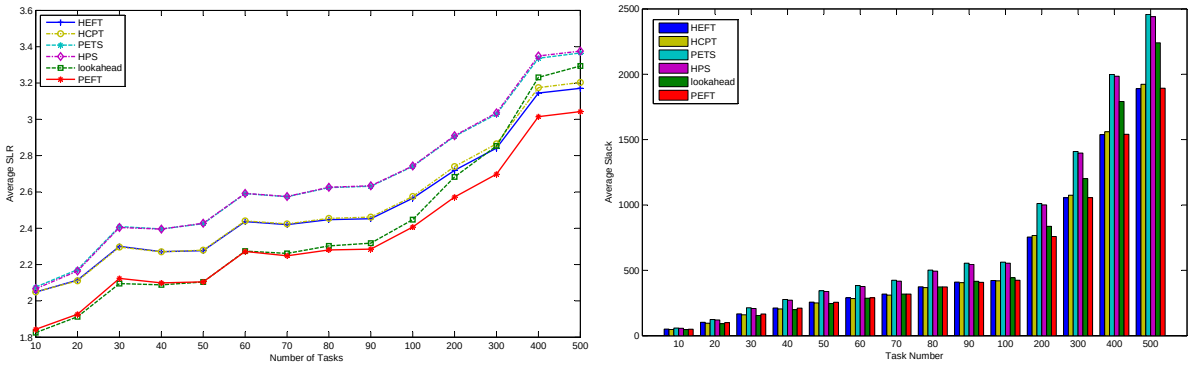
Table 4 shows the percentage of better, equal and worse results for PEFT when compared with the remaining algorithms, based on *makespan*. Compared with HEFT, PEFT achieves better scheduling in 72% of runs, equivalent schedules in 3% of runs and worse schedules in 25% of runs.

		PEFT	Lookahead	HEFT	HCPT	PETS	HPS
PEFT	better	*	66%	72%	79%	91%	90%
	worse		28%	25%	18%	9%	8%
	equal		6%	3%	3%	0%	2%
Look-ahead	better	28%	*	64%	70%	86%	84%
	worse	66%		31%	26%	14%	14%
	equal	6%		5%	4%	0%	2%
HEFT	better	25%	31%	*	29%	91%	72%
	worse	72%	64%		20%	8%	9%
	equal	3%	5%		51%	0%	18%
HCPT	better	18%	26%	20%	*	86%	68%
	worse	79%	70%	29%		14%	13%
	equal	3%	4%	51%		0%	18%
PETS	better	9%	14%	8%	14%	*	39%
	worse	91%	86%	91%	86%		60%
	equal	0%	0%	0%	0%		1%
HPS	better	8%	14%	9%	13%	60%	*
	worse	90%	84%	72%	68%	39%	
	equal	2%	2%	18%	13%	1%	

TABLE 4: Pair-wise schedule length comparison of the scheduling algorithms

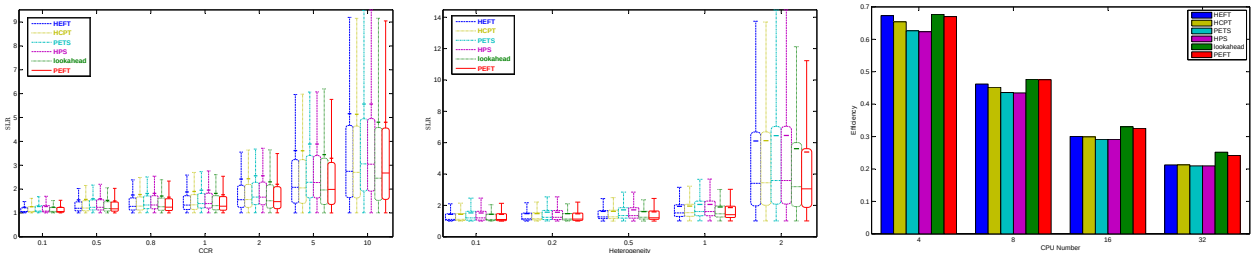
5.3 Real-world application graphs

In addition to the random graphs, we evaluated the performance of the algorithms with respect to real-world applications, namely Gaussian Elimination [2], Fast Fourier Transform [12], [29], Laplace Equation [32], Montage [3], [6], [16] and Epigenomics [4], [10]. All of these applications are well known and used in real-world problems. Because of the known structure of these applications, we simply used different values for CCR, heterogeneity and CPU number. The range of values that we used in our simulation was $[0.1, 0.5, 0.8, 1, 2, 5, 10]$ for CCR, $[0.1, 0.2, 0.5, 1, 2]$ for heterogeneity and $[2, 4, 8, 16, 32]$ for CPU number. For Montage and Epigenomics, we also considered 64 CPUs. The range of parameters considered here represents typical values within the context of this work [15], [29]. The CCR and heterogeneity represent a wide range of machines, from high-speed networks (CCR=0.1) to slower ones (CCR=10) and from nearly homogeneous systems (heterogeneity equal to 0.1) to highly heterogeneous machines (heterogeneity equal to 2). We also considered a wide range of CPU



(a) (b)

Fig. 3: **(a)** Average SLR and **(b)** slack for random graphs as a function of DAG size



(a) (b) (c)

Fig. 4: (a) Boxplot of SLR as a function of CCR; (b) boxplot of SLR as a function of heterogeneity and (c) efficiency for random graphs

numbers to simulate higher concurrent environments with 2 processors, as well as low concurrent situations where the total number of processors, in most of the cases, is higher than the maximum number of concurrent tasks ready to be scheduled at any given instant.

5.3.1 Gaussian Elimination

For Gaussian Elimination, a new parameter, matrix size m, was used to determine the number of tasks. The total number of tasks in a Gaussian Elimination graph is equal to $\frac{m^2+m-2}{2}$. The values considered for m were $\left[5, 10, 15, 20, 30, 50, 100\right]$. The boxplot of SLR as a function of the matrix size is shown in Figure 5.

For all matrix sizes, PEFT produced shorter schedules than all other algorithms with quadratic complexity and almost the same results as the Lookahead algorithm, which has quartic complexity. Figure 6 shows the SLR boxplot as a function of the CCR parameter. For low CCRs, PEFT yielded results equivalent to those of HEFT, and for higher CCRs, PEFT performed significantly better, obtaining an average improvement of 2%, 9% and 16% over HEFT for CCR values of 2, 5 and 10, respectively. From the boxplots, we can see that statistically, PEFT produced schedules with lower SLR dispersion than the other quadratic algorithms.

Concerning heterogeneity, PEFT outperformed HEFT in average SLR by 2%, 3%, 4%, 6% and 7% for heterogeneity values of 0.1, 0.2, 0.5, 1 and 2, respectively. Concerning the number of CPUs, the improvement over HEFT was 2%, 6%, 12% and 12% for sets of 4, 8, 16 and 32 CPUs, respectively. Graphical representation is not provided for SLR as a function of heterogeneity and CPU numbers.

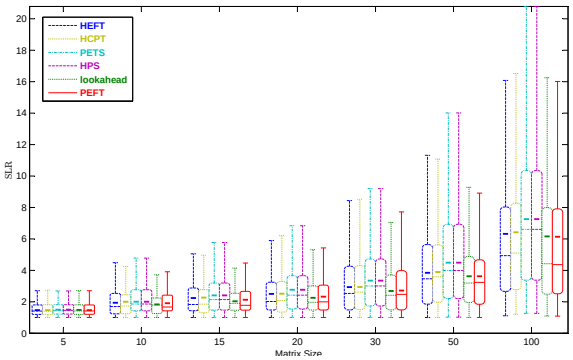


Fig. 5: Boxplot of SLR for the Gaussian Elimination graph as a function of matrix size

5.3.2 Fast Fourier Transform

The second real application was the Fast Fourier Transform (FFT). As mentioned in [29], we can separate the FFT algorithm into two parts: recursive calls and the butterfly operation. The number of tasks depends on the number of FFT points (n), where there

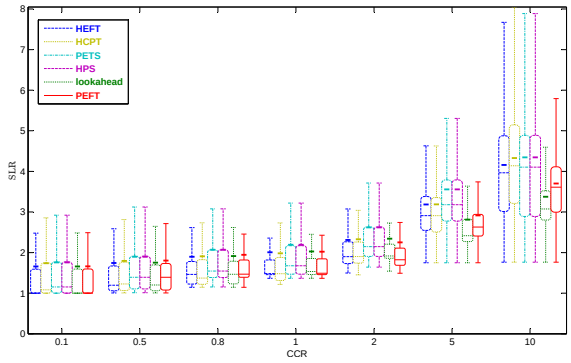


Fig. 6: Boxplot of SLR for the Gaussian Elimination graph as a function of the CCR

are $2 \times (n-1)+1$ recursive call tasks and $n \log_2 n$ butterfly operation tasks. Also, in this application, because the structure is known, we simply change CCR, β and the CPU number. Figure 7 shows the SLR for different FFT sizes.

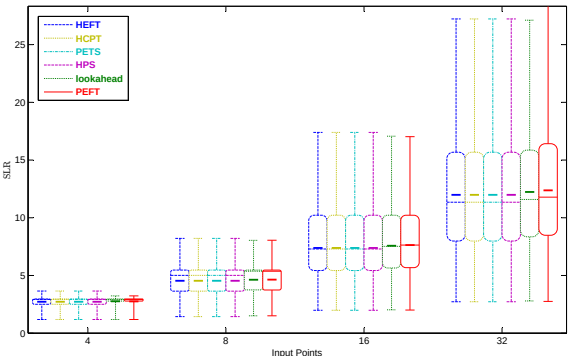


Fig. 7: Boxplot of SLR for the Fast Fourier Transform graph as a function of the input points

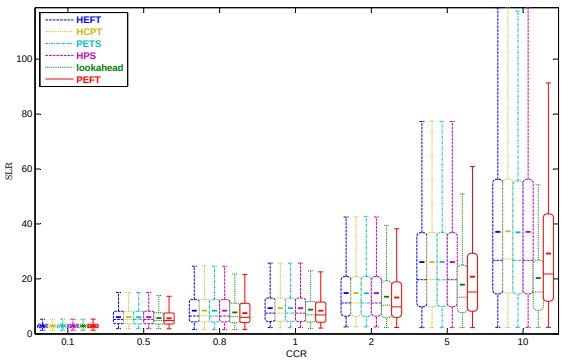
In this type of application, PEFT and Lookahead yielded the worst results, although the results were similar to those obtained using HEFT. This example allowed us to conclude, with additional experiments, that PEFT does not perform better than HEFT when all tasks belong to a critical path, i.e., when all paths are critical.

5.3.3 Montage Workflow

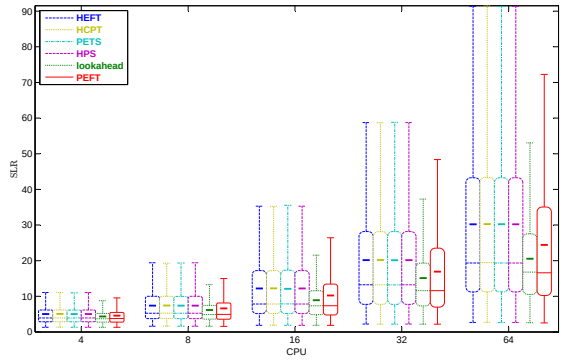
Montage is an application for constructing custom astronomical image mosaics of the sky.

We considered Montage graphs with 25 and 50 tasks, and as in the other real applications, because the graph structure was defined, we simply considered different values of CCR, β and CPU number. Figure 8 shows the boxplots of SLR as a function of different hardware parameters.

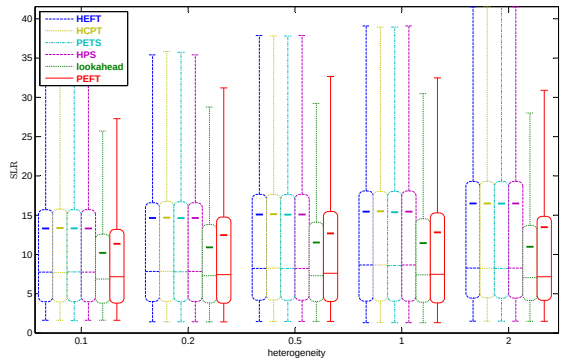
All algorithms except for PEFT and Lookahead exhibited the same performance for this application. The average SLR improvement for PEFT over HEFT for different values of CCR (Figure 8a) started at 0.8% for



(a)



(b)



(c)

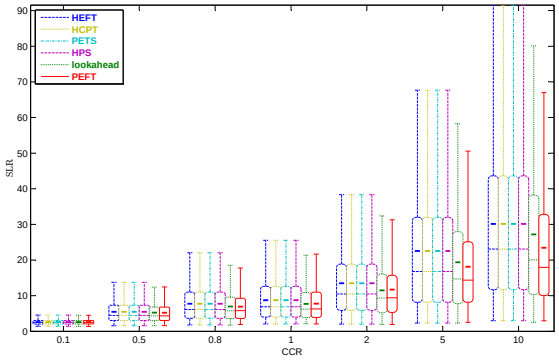
Fig. 8: Boxplot of SLR for Montage with respect to (a) CCR, (b) number of CPUs and (c)heterogeneity factor

a low CCR value (equal to 0.1) and increased to 22% at a CCR value equal to 10. Concerning the number of CPUs, the improvement was 10% with 4 CPUs and increased to 19% for 64 CPUs, as shown in Figure 8b. The improvement for different heterogeneities, Figure 8c, started at 15% for low heterogeneity ($\beta = 0.1$) and increased to 18% for a heterogeneity of 2 ($\beta = 2$).

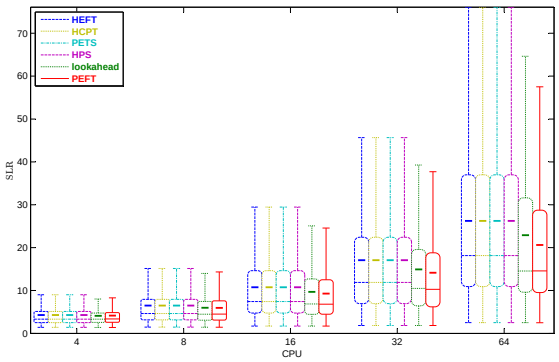
5.3.4 Epigenomic Workflow

The Epigenomic workflow is used to map the epigenetic state of human cells on a genome-wide scale. As was the case for the other real application graphs, the structure of this application is known; therefore,

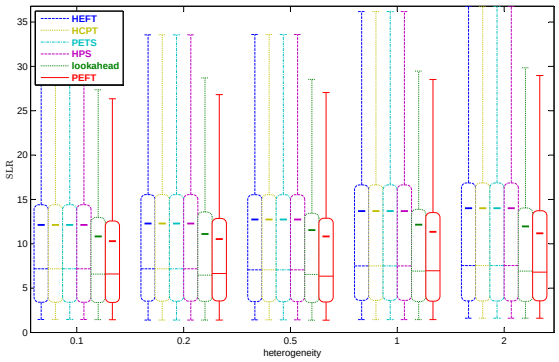
we simply considered different values of CCR, β and CPU number. In our experiment, we used graphs with 24 and 46 tasks.



(a)



(b)



(c)

Fig. 9: Boxplot of SLR for Epigenomic Workflow as a function of (a) CCR, (b) CPU number and (c) heterogeneity factor

Figure 9 shows the boxplot for SLR as a function of the hardware parameters. Also, for this application, PEFT always outperformed the other algorithms, including the Lookahead algorithm. The average SLR improvement of PEFT over HEFT for a low CCR value of 0.1 was 0.1% and increased to 22% for a CCR value of 10 (Figure 9a). Similarly, PEFT showed (Figure 9b) a range of SLR improvement for different CPU numbers: 3% for 4 CPUs and 21% for 64 CPUs. In addition, we observed an average SLR improvement of 15% to 21% for low heterogeneity ($\beta = 0.1$) to high heterogeneity ($\beta = 2$) (Figure 9c).

6 Conclusions

In this paper, we proposed a new list scheduling algorithm with quadratic complexity for heterogeneous systems called PEFT. This algorithm improves the scheduling provided by state-of-the-art quadratic algorithms such as HEFT [29]. To our knowledge, PEFT is the first algorithm to outperform HEFT while maintaining the same time complexity of $O(v^2.p)$. The algorithm is based on an Optimistic Cost Table (OCT) that is computed before scheduling. This cost table represents for each pair (task, processor) the minimum processing time of the longest path from the current task to the exit node by assigning the best processors to each of those tasks. The table is optimistic because it does not consider processor availability at a given time. The values stored in the cost table are used in the processor selection phase. Rather than considering only the Earliest Finish Time (EFT) for the task that is being scheduled, PEFT adds to EFT the processing time stored in the table for the pair (task, processor). All processors are tested, and the one that gives the minimum value is selected. Thus, we introduce the look ahead feature while maintaining quadratic complexity. This feature has been proposed in other algorithms, but all such algorithm increase the complexity to cubic or higher orders.

To prioritize the tasks, we also use the OCT table to define a new rank that is given by the average of the costs for a given task over all processors. Although other ranks could be used, such as $rank_u$ [29], we concluded that with the new rank, similar performance is obtained. Therefore, the use of $rank_u$ would require additional computations without resulting in a significant advantage.

In terms of Scheduling Length Ratio (SLR), the PEFT algorithm outperformed all other quadratic algorithms considered in this work for random graph sizes of 10 to 500. Statistically, PEFT had the lowest average SLR and a lower dispersion in the distribution of the results.

We also compared the algorithms in terms of robustness to uncertainty in the task processing time, given by the *Slack* function, and we obtained the same level of robustness for PEFT and HEFT, which is an important characteristic of the proposed algorithm.

The simulations performed for real-world applications also verified that PEFT performed better then the remaining quadratic algorithms. These tests also revealed an exceptional case (the FFT transform) in which PEFT did not perform better. We concluded that this lack of improvement by PEFT occurs for graphs with the same characteristics as FFT and that are characterized by having all tasks belong to a critical path, i.e., having all paths be critical.

From the results, we can conclude that among the static scheduling algorithms studied in this paper, PEFT exhibits the best performance for the static scheduling of DAGs in heterogeneous platforms with quadratic time complexity and the lowest quadratic time complexity.

REFERENCES

[1] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen and R.F. Freund, "Dynamically mapping of a class of independent tasks onto heterogeneous computing systems", *Journal of Parallel and Distributed Computing*, vol. 59, pp. 107-131, 1999.

[2] A.K. Amoura, E. Bampis and J.C. Konig, "Scheduling algorithms for parallel Gaussian elimination with communication costs", *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 7, pp. 679-686, 1998.

[3] Montage, An Astronomical Image Mosaic Engine, http://montage.ipac.caltech.edu/

[4] USC epigenome center, http://epigenome.usc.edu/

[5] H. Arabnejad and J.G. Barbosa, "Performance Evaluation of List Based Scheduling on Heterogeneous Systems", *Euro-Par 2011: Parallel Processing Workshops*, vol. 7155 of *Lecture Notes in Computer Science*, pp. 440-449, 2012.

[6] G.B. Berriman, J.C. Good, A.C. Laity, A. Bergou, J. Jacob, D.S. Katz, E. Deelman, C. Kesselman, G. Singh, M.-H. Su and R. Williams, "Montage: A grid enabled image mosaic service for the national virtual observatory", *Astronomical Data Analysis Software and Systems (ADASS) XIII*, vol. 314 of *Astronomical Society of the Pacific Conference Series*, pp. 593-596, 2004.

[7] L.F. Bittencourt, R. Sakellariou and E.R.M. Madeira, "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm", *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP'10)*, pp. 27-34, 2010.

[8] C. Boeres, J.V. Filho and V.E.F. Rebello, "A Cluster-based Strategy for Scheduling Task on Heterogeneous Processors" *16th Symposium on Computer Architecture and High Performance Computing*, pp. 214-221, 2004.

[9] L. Bölöni and D.C. Marinescu, "Robust scheduling of metaprograms", *Journal of Scheduling*, vol. 5, no. 5, pp. 395-412, 2002.

[10] D.A. Brown, P.R. Brady, A. Dietz, J. Cao, B. Johnson and J. McNabb, "A case study on the use of workflow technologies for scientific analysis: Gravitational wave data analysis", *Workflows for e-Science*, pp. 39-59, 2007.

[11] L.C. Canon, E. Jeannot, R. Sakellariou and W. Zheng, "Comparative evaluation of the robustness of dag scheduling heuristics", in Sergei Gorlatch, Paraskevi Fragopoulou and Thierry Priol, *Grid Computing - Achievements and Prospects*, pp. 73-84, Springer, 2008.

[12] Y.C. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors", *Proceedings Supercomputing'92*, pp. 512-521, 1992.

[13] B. Cirou and E. Jeannot, "Triplet: A clustering scheduling algorithm for heterogeneous systems", *International Conference on Parallel Processing Workshops*, pp. 231-236, 2001.

[14] Edward G. Coffman, "Computer and Job-Shop Scheduling Theory", John Wiley & Sons Inc, 1976.

[15] M.I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems", *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 399-409, 2008.

[16] E. Deelman, G. Singh, M.H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob and D.S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems", *Journal of Scientific Programming*, vol. 13, no. 3, pp. 219-237, IOS Press, 2005.

[17] H. El-Rewini and T.G. Lewis, "Scheduling Parallel Program Tasks onto Arbitrary Target Machines", *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138-153, 1990.

[18] M. R. Garey and D. S. Johnson, "Computers and intractability; a guide to the theory of *NP*-completeness", W.H. Freeman, 1979.

[19] T. Hagras and J. Janeček, "A Simple Scheduling Heuristic for Heterogeneous Computing Environments", *Proceedings Second International Symposium on Parallel and Distributed Computing*, pp. 104-110, 2003.

[20] T. Hagras and J. Janeček, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems", *Journal of Parallel Computing*, vol. 31, no. 7, pp. 653-670, 2005.

[21] E. Ilavarasan and P. Thambidurai, "Low complexity performance effective task scheduling algorithm for heterogeneous computing environments", *Journal of Computer sciences*, vol. 3, no. 2, pp. 94-103, 2007.

[22] E. Ilavarasan and P. Thambidurai and R. Mahilmannan, "High performance task scheduling algorithm for heterogeneous computing system", *Distributed and Parallel Computing*, vol. 3719 of *Lecture Notes in Computer Science*, pp. 193-203, 2005.

[23] M.A. Iverson, F. Özgüner and G.J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment", *4th Heterogeneous Computing Workshop (HCW'95)*, pp. 93-100, 1995.

[24] H. Oh and S. Ha, "A Static Scheduling Heuristic for Heterogeneous Processors", *Euro-Par'96 Parallel Processing*, vol. 1124 of *Lecture Notes in Computer Science*, pp. 573-577, 1996.

[25] A. Radulescu and A.J.C. van Gemund, "Fast and Effective Task Scheduling in Heterogeneous Systems", *9th Proceedings Heterogeneous Computing Workshop (HCW)*, pp. 229-238, 2000.

[26] Z. Shi, E. Jeannot and J.J. Dongarra, "Robust task scheduling in non-deterministic heterogeneous computing systems", *IEEE International Conference on Cluster Computing*, pp. 1-10, 2006.

[27] G.C. Sih and E.A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architecture", *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175-187, 1993.

[28] F. Suter, "DAG Generation Program", http://www.loria.fr/~suter/dags.html, 2010.

[29] H. Topcuoglu, S. Hariri and M. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260-274, 2002.

[30] H. Topcuoglu, S. Hariri and M. Wu, "Task scheduling algorithms for heterogeneous processors", *8th Proceedings Heterogeneous Computing Workshop (HCW)*, pp. 3-14, 1999.

[31] J.D. Ullman, "NP-complete scheduling problems", *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384-393, 1975.

[32] M.Y. Wu and D.D. Gajski, "Hypertool: A Programming Aid for Message-Passing Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330-343, 1990.