2022-05-30

Workflow Simulation and Multi-Threading Aware Task Scheduling for Heterogeneous Computing

Kelefouras, Vasileios

10.1016/j.jpdc.2022.05.011
Journal of Parallel and Distributed Computing (Elsevier)
Elsevier

Workflow Simulation and Multi-Threading Aware Task Scheduling for Heterogeneous Computing

Vasilios Kelefouras, Karim Djemame

Abstract

Efficient application scheduling is critical for achieving high performance in heterogeneous computing systems. This problem has proved to be NP-complete even for the homogeneous case, heading research efforts in obtaining low complexity heuristics that produce good quality schedules. Such an example is HEFT, one of the most efficient list scheduling heuristics in terms of makespan and robustness.

In this paper, we propose two task scheduling methods for heterogeneous computing systems that can be integrated to several task scheduling algorithms. First, a method that improves the scheduling time (the time for obtaining the output schedule) of a family of task scheduling algorithms is delivered without sacrificing the schedule length, when the computation costs of the application tasks are unknown. Second, a method that improves the scheduling length (makespan) of several task scheduling algorithms is proposed, by identifying which tasks are going to be executed as single-threaded and which as multi-threaded implementations, as well as the number of the threads used. We showcase both methods by using HEFT popular algorithm, but they can be integrated to other algorithms too, such as HCPT, HPS, PETS and CPOP.

The experimental results, which consider 14580 random synthetic graphs and five real world applications, show that by enhancing HEFT algorithm with the two proposed methods, significant makespan gains and high scheduling time gains, are achieved.

Keywords: Task Scheduling, Heuristics, Multithreading, HEFT, Heterogeneity, multi-core, Scheduling Time, Makespan

1. Introduction

Heterogeneous Computing Systems (HCS) offer important benefits over homogeneous systems in various areas such as performance, power consumption, cost etc, leading research efforts on overcoming the challenges that heterogeneity brings at all levels. One of the challenges is the efficient application scheduling, which is the topic of this paper.

A popular representation of an application in this context is the Directed Acyclic Graph (DAG), which includes the characteristics of the application program, such as the application tasks, their computation costs, the data transfer time between tasks and task dependencies [1] [2] [3]. The computation costs can be found by simulation, emulation or by running the tasks on the processors; for the rest of this paper, we will use the word simulation.

The objective of the Task Scheduling (TS) problem is to map the DAG's tasks on the (co)-processors and order their execution so that task precedence requirements are satisfied and a minimum schedule length (aka makespan) is obtained (for the reminder of this paper we will refer to both processors and coprocessors as processors). This problem has proven to be NP-complete [4], even for the homogeneous case, heading research efforts on obtaining low-complexity heuristics that produce good schedules [1], which is the topic of this paper. Such an example is the Heterogeneous Earliest Finish Time (HEFT) [2] algorithm, which is considered one of the most efficient list scheduling algorithms in terms of makespan and robustness [1].

In this paper, we propose two TS methods for HCS and DAG-based applications that can be integrated to several TS algorithms. We showcase both methods by using HEFT algorithm, as it is one of the most efficient list scheduling heuristics in terms of makespan and robustness [1]. In our future work we are planning to apply and evaluate our methods to other algorithms too such as HCPT [5], HPS [6], PETS [3], CPOP [2] and others. Note that this work is an extension of the conference paper in [7].

Our first method, entitled 'Task Scheduling method Reducing the number of task Simulations' (TSRS), reduces the scheduling time of HEFT when the computation costs are unknown. TSRS reduces the number of computation costs required by HEFT and therefore, the number of simulations required/performed, without sacrificing the length of the output schedule. Instead of simulating/running all tasks on every processor (to generate the DAG's computation costs) and then schedule the tasks (by using HEFT), we combine these two phases using an iterative approach; the generation of

the DAG's computation costs and the scheduling of the tasks are applied together, in an iterative approach. First, the DAG is generated whose computation costs refer to one core of the reference processor only. Then, the proposed method is applied which extends the processor selection phase of the algorithm used, in this case HEFT. In the new processor selection phase, we identify the processors which cannot minimize the specific heuristic cost function used, for the current task (let t), regardless of their computation costs; these processors are never selected for t by the algorithm and therefore there is no reason t to be simulated on those processors, reducing the number of simulations performed.

Our second method, 'Multi-threading Effective Task Scheduling' (METS), provides low-complexity heuristics for HCS to find which tasks are going to be executed as Single-Threaded (ST) and which as Multi-Threaded (MT) CPU implementations, as well as the number of the threads used. We show that HEFT's performance is improved without increasing its time complexity (for large DAGs). The application tasks are assumed moldable [8] with the restriction that tasks can only be allocated to the physical cores of one CPU only; moldable tasks are the tasks that can be executed by more than one processors but the number of processors is fixed before execution and stays unchanged afterwards; Pthreads and OpenMP programs are typical examples of moldable tasks [9].

The contributions of this paper are the following:

- A TS method (TSRS) reducing the scheduling time of HEFT popular algorithm, when the computation costs are unknown.
- A low-complexity TS method (METS) improving the scheduling length of HEFT.
- TSRS and METS can be combined reducing both the scheduling time and length.

The evaluation of the proposed methods includes 14580 random synthetic DAGs as well as five real world applications. The experimental results show that TSRS provides simulation gain values from x1.34 to x3.11, while METS provides makespan gains from x1.1 up to x2.3, over HEFT. By combining TSRS with METS, both improved schedule lengths (average speedup of x1.12), and scheduling time (from x4.5 up to x24 fewer simulations), are achieved.

The reminder of this paper is organized as follows. In Section 2, we introduce the TS problem. In Section 3, the related work is reviewed. The proposed methods are given in Section 4, while the experimental results are discussed in Section 5. Finally, Section 6 is dedicated to conclusions.

2. Task Scheduling Formulation

Resource model: The hardware (HW) platform consists of a fixed set of p heterogeneous devices with diverse computation capabilities. The multi-core CPUs are treated as m-core devices, where $m = [1, cores]$ and $cores$ is the number of the physical CPU cores.

Workflow model: A workflow application is modeled as a DAG, $G = (V, E)$, where V is the set of u nodes and each node $u \in V$ represents an application task, which includes instructions that must be executed on the same processor. E is the set of e communication edges between tasks; each $e(i, j) \in E$ represents the task-dependence constraint such that task t_i should complete its execution before task t_j is started [1]. The $n \times p$ computation cost matrix W stores the computation costs of the tasks, where n is the number of the tasks and p is the number of the processors; each element $w_{t,j} \in W$ refers to the estimated time to execute task t on processor p_j (note that in the next paragraphs matrix W becomes $n \times p \times cores$, as multi-threaded implementations exist). The W values can be found by simulation, emulation or by running the tasks on the HW; for the rest of this paper, we will use the word simulation. The execution of any task is considered nonpreemptive.

Each edge $e(i, j) \in E$ is associated with a non-negative weight value $d_{i,j}$ that represents the amount of data to be transmitted from task t_i to task t_j. The data transfer rate between any two processors on the network is assumed to be fixed and constant [10]. The communication cost of an edge (t_i, t_j) equals to the amount of data transmitted from task t_i to task t_j , or $d_{i,j}$, divided by the data transfer rate of the network. Since the data transfer rate of the intra-processor bus is much higher than the data transfer rate of the interprocessor network, the communication cost between two tasks scheduled on the same processor is taken as zero. These model simplifications are common in this scheduling problem [1] [2] [10].

TSRS problem definition: This problem is the static scheduling of a single application, whose computation cost matrix W is unknown, in a set of p heterogeneous devices, in such a way that both the scheduling length and the scheduling time (to deliver the output schedule), are minimized. It is important to note that the scheduling time highly depends on the time needed to simulate the tasks and get their computation costs.

Standalone TSRS assumes rigid (non-moldable) tasks, i.e., each task is executed by one only processor. Monotonic computation costs are assumed.

Definition 1. *The notion of monotonic computation costs is defined as follows. Consider a task t_1 and two different processors (p_1, p_2). If $(w_{t_1,p_2} \geq w_{t_1,p_1})$ for task t_1, then we assume that $(w_{t,p_2} \geq w_{t,p_1})$ for every task t.*

METS problem definition: This problem is the static scheduling of a single application, consisting of a set of n moldable tasks, in a set of p heterogeneous devices, in such a way that the scheduling length is minimized. The application tasks are assumed moldable [8] (a single task can be executed by more than one processors) with the restriction that tasks can be allocated to the physical cores of one CPU only; moldable tasks are the tasks being allocated to a fixed number of processors before execution and stay unchanged afterwards. OpenMP programs typify moldable tasks as users can specify the number of the threads before the execution of a parallel program. Thus, given a multi-core CPU with *cores* physical cores, we consider every task as an m-threaded implementation, where $m = [1, cores]$. The computation cost matrix W becomes $n \times p \times cores$; if the processor is not a multi-core CPU (e.g., GPU or single-core CPU), $m = 1$ $(w_{t,j,1})$. The CPU core utilization factor is defined as, $factor_{t,j,m} = w_{t,j,1}/w_{t,j,m}$. Unlike TSRS, the computation cost matrix W is known and therefore the previous assumption about monotonic costs is not applied here.

TSRS+METS problem definition: In this paper, METS is applied together with TSRS, in order to optimize for both scheduling time and length. This problem is the static scheduling of an application consisting of a set of n moldable tasks, whose computation cost matrix W is unknown, in a set of p heterogeneous devices, in such a way that both the scheduling time and scheduling length, are minimized. We make the following assumptions which are common in moldable tasks [8] [9]: a) $w_{t,i,f1} \leq w_{t,i,f2}$, where $f1 \geq f2$, $f1 \leq cores$, and, b) every task scales equally in different CPUs $(factor_{t,i,f} = factor_{t,j,f})$. Thus, if $w_{t,i,1} \leq w_{t,j,1}$, then $w_{t,i,f} \leq w_{t,j,f}$ but $w_{t,i,f1} \not\leq w_{t,j,f2}$, where $f1 < f2$.

Next, we present some common attributes used in TS problem [1] [2] [10], which we will refer to in the following sections.

Definition 2. *$pred(t_i)$ denotes the set of immediate predecessors of task t_i in a given DAG [2].*

Definition 3. *makespan or schedule length denotes the finish time of the last task in the scheduled DAG [2] and is defined as:*

$$makespan = \max\{AFT(t_{exit})\} \tag{1}$$

where $AFT(t_{exit})$ denotes the Actual Finish Time of the exit task.

Definition 4. *$EST(t_i, p_j, m)$ denotes the Earliest Start Time (EST) of task t_i on processor p_j using m threads and each thread is mapped on a specific CPU core (if p_j is not a CPU, m=1) and defined as*

$$EST(t_i, p_j, m) = \max\left\{T_{Avail}(p_j, m), T_{pred}(t_i, p_j)\right\}$$
$$T_{pred}(t_i, p_j) = \max_{t_l \in pred(t_i)}\{AFT(t_l) + c_{l,i}\} \quad (2)$$

where $T_{Avail}(p_j, m)$ is the earliest time at which the m cores (that the m threads run) of processor p_j are ready and $T_{pred}(t_i, p_j)$ is the time at which all data needed by task t_i arrive at the processor p_j. The communication cost $c_{l,i}$ is zero if the predecessor task t_l is assigned to processor p_j. For the entry task, $EST(t_{entry}, p_j, m) = 0$.

Definition 5. *$EFT(t_i, p_j, m)$ denotes the Earliest Finish Time (EFT) of a task t_i on processor p_j using m threads:*

$$EFT(t_i, p_j, m) = EST(t_i, p_j, m) + w_{t,j,m} \quad (3)$$

which is the EST of a task t_i on the m cores of processor p_j, plus the computation cost of the m-threaded implementation of t_i on processor p_j. For the rest of this paper we will refer to $EFT(t_i, p_j, 1)$ as $EFT(t_i, p_j)$.

3. Related Work and Background Knowledge

TS can be performed at compile-time or at run-time, referred as static or dynamic scheduling. In the static scheduling case, all the information regarding the application and computing resources is assumed available a priori. In the dynamic case, such information is not available and decisions are made at runtime. A taxonomy of all the task mapping methodologies (both static and dynamic) is given in [11].

The static TS algorithms are classified in two main categories. The first one includes algorithms that are based on heuristics, such as list scheduling [1] [2], clustering [12] or node duplication [13], while the second includes stochastic search algorithms, where the problem is modelled as an optimization problem using either ILP [14], CP models [15] or hybrid ILP-CP models along with advanced algorithms in order to reduce the simulation time [16]. Clustering heuristics are mainly proposed for homogeneous systems [12]. The duplication heuristics produce shorter makespans than list scheduling heuristics, but result in higher time complexity and more processor availability and power [1]. List scheduling heuristics, on the other hand, produce the most efficient schedules, without compromising the makespan and with a low complexity [1]. Some of the most important list scheduling heuristics for

heterogeneous systems are: PEFT [1], HEFT [2], HCPT [5], HPS [6],PETS [3], Lookahead [17], MOHEFT [18], LDCP [10], SDBATS [19], DVR HEFT [20], LB-HEFT [21], HOFT [22]. In [23], HEFT is modified to consider a budget limit for the hourly-based cost model of modern Infrastructure as a Service (IaaS) clouds. In [24], HEFT is integrated with a fuzzy dominance sort mechanism in order to optimize both the cost and the makespan in IaaS clouds. [25] and [26] propose HEFT hybrid variants for HCS.

HEFT algorithm assumes rigid tasks (non-moldable) and is shown in Algorithm 1; it has a prioritizing and a processor selection phase. In the first phase, task priorities are defined by using $rank_u$ which represents the cost of the longest path from t_i to the exit node, including the computation cost of t_i and is given by $rank_u(t_i) = \overline{w_i} + \max_{t_j \in succ(t_i)}\{\overline{c_{(i,j)}} + rank_u(t_j)\}$. The bar over numbers indicates mean values. For the exit task, $rank_u(t_{exit}) = \overline{w_{exit}}$. The task list is ordered by decreasing value of $rank_u$. The task with the highest rank is scheduled first. In the processor selection phase, the task with the higher $rank_u$ value is assigned to the processor giving the EFT.

Algorithm 1 HEFT Algorithm

1: Set the computation costs of tasks and communication costs of edges with mean values
2: Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task
3: Sort tasks in a scheduling list by decreasing order of $rank_u$ values
4: **while** there are unscheduled tasks in the list **do**
5: Select the first task, t_i, from the list for scheduling
6: **for** each processor p_j in the processor-set **do**
7: Compute $EFT(t_i, p_j)$ value using the insertion-based scheduling policy
8: **end for**
9: Assign task t_i to the processor p_j that minimizes EFT of task t_i
10: **end while**

All the aforementioned algorithms take as input a DAG containing the computation and communication costs and therefore the quality of the output schedule is affected by the DAG values too. The tasks' execution time estimation problem is not as well-developed as the scheduling problem, because several straightforward techniques exist which provide acceptable performance [27]. Regarding the popular list scheduling algorithms such as HEFT, HCPT, HPS, PETS etc, they consider the execution time of a task on a processor, as a constant value. However, there are algorithms that a) don't use the execution time values of the tasks, but instead use a list of tasks ordered by their execution time [28], b) use knowledge of the expected execution time value and the variance in order to measure the uncertainty of

the workflow execution time [29], c) use the worst case execution time values (WCET), d) use the execution time estimates as random variables instead of keeping them constant, and provide complete information about them at maximum precision [27].

To the best of our knowledge there is no similar work to TSRS, addressing the static TS problem and the problem of generating the computation costs, together.

On the other hand METS is close to the problem of scheduling moldable tasks with the restriction that tasks can only use the cores of one CPU [8] [9]; most of the existing works are based on a two-phase approach. First, the number of processors assigned for each task is selected and second, the rigid (non-moldable) tasks are scheduled by using a TS algorithm. In [8], they present a new algorithm combining dual approximation and ILP for moldable tasks on hybrid platforms of identical GPUs and CPUs. In [9], a new algorithm is proposed that supports arbitrary run-time functions of moldable tasks on identical processors. In [30], a scheduling algorithm with a tunable performance guarantee is developed, for homogeneous multicluster platforms. An extensive comparison of several TS algorithms for moldable tasks is carried out in [31], for identical processors (both theoretical and experimental). [32] employs ILP for streaming applications while [33] employs ILP considering inter intra task communications. Last, in [34] they present efficient algorithms for scheduling an application on hybrid platforms of identical CPUs and GPUs. Comparing to the aforementioned methods, METS achieves lower time complexity and is applicable to HCS. Nevertheless, in this paper, METS is applied together with TSRS and not as a standalone method and thus the moldable problem is addressed without requiring all the computation costs in the DAG, further reducing the scheduling time.

SKOPE [35] is a framework that produces a descriptive model about the semantic behaviour of a workload. StarPU [36] provides designers with a convenient way to execute parallel tasks over heterogeneous hardware and tune scheduling algorithms. The Pegasus Workflow Management System [37] is a framework for mapping complex scientific workflows onto distributed resources. A technique to reduce the number of simulations needed during system-level design space exploration is proposed in [38]. SimSo [39] is a simulation tool that facilitates the comparison of different schedulers. The SESAME [40] framework, which is part of the Daedalus framework [41], provides modeling and simulation methods and tools for the efficient design space exploration of heterogeneous embedded multimedia systems. In

[42] [43], novel methods scheduling the co-running threads in multi-core platforms are proposed. In [44], a novel model-based data partitioning algorithm is proposed.

4. Proposed TS method & Heuristics

In this section we introduce two novel Task Scheduling (TS) methods for heterogeneous computing systems (HCS), a TS method Reducing the number of task Simulations performed (TSRS) which is given in Subsection 4.1 and Multi-threading Effective Task Scheduling (METS) heuristics which are given in Subsection 4.2. In Subsection 4.3, TSRS and METS are combined.

4.1. Task Scheduling method Reducing the number of task Simulations (TSRS)

In Algorithm 2, we show how TSRS is applied to HEFT. TSRS consists of two stages, an initialization stage (line 1 in Algorithm 2), where all the processors are sorted in an increasing computational capability (CC) order and the main stage (line 6 in Algorithm 2). The main stage of TSRS extends/modifies the processor selection phase, lines 6-8 in Algorithm 1. As it can be observed, Algorithm 1 and Algorithm 2 differ only in line 1 and lines 6,7, where the initialization and main stage are performed, respectively.

Algorithm 2 HEFT with TSRS / HEFT with TSRS+METS

1: Sort in an increasing order all the groups of processors according to their computation capability (CC). Set the computation costs of tasks according to p_{ref} only ($w_{t,p_{ref},1}$) and the communication costs of edges with mean values
2: Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task
3: Sort tasks in a scheduling list by decreasing order of $rank_u$ values
4: **while** there are unscheduled tasks in the list **do**
5: Select the first task, t_i, from the list for scheduling
6: [$w_{t_i,i,1}$(), SL()]=TSRS(t_i); / [$w_{t_i,i,m}$(), SL()]=TSRS+METS(t_i);
7: **for** each processor p_j in SL (simulation list) **do**
8: Compute $EFT(t_i, p_j)$ / $EFT(t_i, p_j, m)$ value with/without the insertion-based scheduling policy
9: **end for**
10: Assign task t_i to the processor p_j that minimizes EFT of task t_i
11: **end while**

In line 1 (Algorithm 2), the DAG is initialized with the computation costs of the tasks on the one core of p_{ref} only (reference processor), i.e., $w_{t,p_{ref},1}$. Furthermore, all the processors are classified into groups, according to their computation capability (CC); a random task is run on every processor and the execution time values are measured. Then, all the processors are classified into groups according to the execution time values measured. In the case

where the execution time values of the random task on two different processors is approximately the same, we can consider both processors in the same processor group. All the groups are sorted in an increasing CC order, e.g., $proc\_order = (p_{type2}, p_{type3}, p_{type1})$. The processor achieving the minimum execution time value is considered as the one with the highest CC; regarding multi-core CPUs, the CC refers to one core only (ST implementations).

The generation of the other computation costs and the scheduling of the tasks are applied together, in an iterative approach; in line 6, TSRS discards the processors which cannot minimize the specific heuristic cost function used for the current task (regardless of their computation costs), while all the others are simulated and their computation costs are returned.

The DAG is initialized with the computation costs on the reference processor (p_{ref}) only and therefore the $rank_u$ values are no longer computed using the average costs but using the computation costs on p_{ref}, slightly affecting the task priority list; the priority list is not substantially affected because the computation costs are assumed monotonic. In [45], the rank function of HEFT algorithm is investigated by using the mean, median, worst and best computation costs; it is shown that for random computation costs (not monotonic as in our case) first, different ways of computing $rank_u$ affects HEFT performance and second, the mean computation costs is not the best choice. In Subsection 5.3, we show that HEFT's schedule length is not degraded by TSRS and in addition to [45], we showcase that the mean computation costs do not provide better solutions than the p_{ref} ones. In terms of makespan, it is more efficient to select a Highest Computational Capability Processor (HCCP) as p_{ref} (a last group's processor). However, in METS (Subsection 4.3), p_{ref} cannot be a HCCP in all cases, because it has to be the multi-core processor containing the maximum number of cores (must be a CPU). Thus, given that TSRS is applied as both standalone method and together with METS, we will not consider p_{ref} as a fixed value.

The main step of TSRS (line 6 in Algorithm 2) is given by Subsection 4.1.2 and Subsection 4.1.1, when the insertion based scheduling policy is used or not, respectively.

4.1.1. TSRS without insertion based scheduling policy

The main step of TSRS reduces the number of candidate processors in the processor selection phase. The procedure follows. The EFT is given by Eq. 3 and consists of two parts, EST and $w_{i,j,f}$. The second part of Eq. 3 ($w_{i,j,f}$) is an unknown value, as task t is not simulated on every processor group but on

p_{ref} only, while the first part of Eq. 3 is known, as it refers to the processor availability time as well as to the finish time of the previously scheduled tasks. Given that first, the processor groups are sorted in an increasing CC order and second, the first part of Eq. 3 is known, we are able to reduce the number of candidate processors for task t, without excluding any processor with minimum EFT value. As an example, assume that the EFT values of t on 4 different single core processors are those in Eq. 4 and also $p_{ref} = p_3$.

$$\begin{aligned} EFT(t, p_1) &= w_{t,1,1} + 10. \\ EFT(t, p_2) &= w_{t,2,1} + 9. \\ EFT(t, p_3) &= 2 + 9. \\ EFT(t, p_4) &= w_{t,4,1} + 13. \end{aligned} \tag{4}$$

Given that $(w_{t,1,1} \geq w_{t,2,1} \geq w_{t,3,1} = 2 \geq w_{t,4,1})$, there is no need to simulate t on p_1 and p_2 as these two processors always give a larger EFT value than p_3 and therefore they will never be allocated for t by HEFT.

Algorithm 3 TSRS without using the insertion based scheduling policy

```
[w_{t,i,1}(), SL()] = TSRS (t) {
//step1. Compute the EFT values
for (i = 1, Proc.groups) do
    compute EFT(t, j) for every p_j in group i, by using w_{t,i,1} = w_{t,p_ref,1}
    Put the min EFT(t, j) value from every processor group i in S(i)
end for
//step2. Reduce the search space
Put all processor groups in the simulation list (SL)
for (i = Proc.groups, 2, −1) do
    for (j = i − 1, 1, −1) do
        if (S(i) ≤ S(j)) then
            remove processor group j from SL
        end if
    end for
end for
//step3. this step is optional
if (p_ref ∉ HCCP group) then
    for (i = 1, Proc.groups − 1) do
        if (S(i) ≤ min_EFT_on_p_HCCP) then
            remove p_HCCP group from SL
        end if
    end for
end if
Get the w_{t,i,1} values that i ∈ SL (if any) //t is simulated
Return w_{t,i,1}(), SL() }
```

The proposed method is given in Algorithm 3. In step1, we compute the EFT values for all the processors by using $w_{t,p_{ref},1}$ instead of $w_{t,j,1}$ and put the minimum EFT value of every processor group i in $S(i)$ (lines 3-6 in Algorithm 3). All the processors inside a group have identical computation costs, i.e., $w_{t,i,1}$.

In step2, we compare $S(i)$ with $S(j)$, where always holds $(i > j)$ (and therefore $w_{t,i,1} \leq w_{t,j,1}$). If the $EFT(t,i)$ value referring to processor group i is smaller or equal to any other $EFT(t,j)$ value to a slower group j, then j is not a candidate group and it is removed from the simulation list (SL). Let us follow the above example of Eq. 4, where $EFT(t,p_1) = 12$, $EFT(t,p_2) = 11$, $EFT(t,p_3) = 11$, $EFT(t,p_4) = 15$). First, the $EFT(t,p_4)$ value is compared to $EFT(t,p_3)$,$EFT(t,p_2)$ and $EFT(t,p_1)$ but the *if-condition* in line 13 is never true. Then, the $EFT(t,p_3)$ value is compared to $EFT(t,p_2)$ and $EFT(t,p_1)$ and because $EFT(t,p_2)$ and $EFT(t,p_1)$ give larger or equal values, they are both excluded from SL etc. Thus, the processor groups with $j = 1$ and $j = 2$ are removed from the list. The number of candidate processors is reduced without excluding any processors with minimum EFT value.

In case that ($p_{ref} \in HCCP$ group), step3 (Algorithm 3) is not needed. On the other hand, when p_{ref} is not a HCCP, the method given in step3 (Algorithm 3) is not able to reduce the number of simulations on the HCCP group. To do so, we have to define a lower bound value regarding how fast the HCCP is. We can define a very low unreachable lower bound value on the HCCP, e.g., task t will never run 50 times faster than p_{ref} ($w_{t,p_{ref},1}/50 \leq w_{t,p_{HCCP},1} \leq w_{t,p_{ref},1}$) for every task t. This procedure is given in step3; if $S(i)$ (where $i < Proc.groups$ - $Proc.groups$ is the HCCP group) is lower or equal to the minimum $EFT(t,HCCP)$ value that the HCCP group can get, then the HCCP group is removed from the SL. Let us follow the previous example (Eq. 4), where step2 has already excluded p_1 and p_2 from the SL. If we apply step3 with ($min\_EFT\_on\_p_{HCCP} = 2/50 + 13$), then the minimum value that p_4 can get is always larger than ($EFT(t,p_3) = 11$) and thus p_4 is also excluded from SL.

However, step3 slightly degrades HEFT's output schedule length because the $w_{t,p_{ref},1}$ value is used instead of the real simulation time values ($w_{t,i,1}$) and we do not know how larger the $w_{t,i,1}$ values can be in comparison with $w_{t,p_{ref},1}$. Let us give an example, consider we have to compute the EFT values of t on 4 different single core processors and p_3 is the p_{ref}. Moreover, consider that Eq. 3 gives the following values:

$$\begin{aligned} EFT(t, p_1) &= w_{t,1,1} + 9. \\ EFT(t, p_2) &= w_{t,2,1} + 9. \\ EFT(t, p_3) &= 2 + 15. \\ EFT(t, p_4) &= w_{t,4,1} + 13. \end{aligned} \tag{5}$$

In this case, step2 will exclude p_1 (when $S(2)$ is compared to $S(1)$) and p_3 (when $S(4)$ is compared to $S(3)$) from the SL. In step3, ($min\_EFT\_on\_p_{HCCP}$ = 2/50+13), and thus p_4 is excluded from the SL, as $S(2) \leq min\_EFT\_on\_p_{HCCP}$, meaning that t is assigned on p_2, which is not always the processor with the minimum EFT (it depends on the $w_{t,2,1}$ value); we know that ($w_{t,2,1} \geq 2$), but we don't know how large $w_{t,2,1}$ is; thus, if ($w_{t,2,1} + 9 > (2/50 + 13)$) and therefore ($w_{t,2,1} > 2/50 + 4$), then t may run faster on p_4 than on p_2, and in that case, it shouldn't have been removed from the list. In Subsection 5.3, we show that the more the processor groups, the more the makespan degradation. However, the above refer to special cases only and therefore the makespan degradation is very low.

At last, t is simulated for the processors in SL only (line 27) and the computation costs are returned (line 28).

The time complexity of Algorithm 3 is $O(y^2)$, where y is the number of the processor groups; the maximum value of y is the number of the processors. Thus, TSRS slightly increases HEFT's complexity from $O(e \times p)$ to $O(e \times p^2)$, where e and p is the number of the tasks and processors, respectively. Nevertheless, the algorithm's complexity is undermined as the generation of the DAG computation costs and the scheduling of the tasks are applied together, in an iterative approach, and the time needed for generating the DAG is much higher.

4.1.2. TSRS with insertion based scheduling policy

Some of the task scheduling algorithms, including HEFT, HPS and PETS, compute the EFT value using the insertion based scheduling policy; this policy considers the possible insertion of a task in an earliest idle time slot between two already-scheduled tasks on a processor. The length of an idle time-slot, i.e., the difference between execution start time and finish time of two tasks that were consecutively scheduled on the same processor, should be larger or equal to the computation cost of the task to be scheduled, i.e., ($slot \geq w_{t,i,1}$). Additionally, scheduling on this idle time slot should preserve the precedence constraints, i.e., $T_{pred}(t, i) < T_{avail}(i)$ and $slot \in [T_{pred}(t, i), T_{avail}(i))$.

The above policy needs the computation costs of all processors. The key idea of our method to overcome this problem is that although we do not know $w_{t,i,1}$ value for every i, we use $w_{t,p_{ref},1}$ value to find out whether a possible slot exists or not. In particular, we leverage the fact that if there is no available slot for the minimum $w_{t,i,1}$ value, then there is no slot for any value as ($slot \geq w_{t,i,1}$) and therefore the computation costs are not needed. On the other hand, if there is an available slot for the minimum $w_{t,i,1}$ value, there is no guarantee that there is a slot for any $w_{t,i,j}$ value. In case there is an available slot for group i and i belongs to the SL, only then we simulate t on i, get the $w_{t,i,1}$ value and find out whether the slot truly exists or not and when. The number of simulations performed is higher than Algorithm 3.

The procedure is given in Algorithm 4. In lines 4-8 we find the minimum $w_{t,i,1}$ value for i. Regarding $i < p_{ref}$, always $w_{t,p_{ref},1} \leq w_{t,i,1}$ (the groups are sorted). Regarding $i > p_{ref}$, we use a very low computation cost value that will never be reached, e.g., $w_{t,p_{ref},1}/50$. In line 10, $T2$ variable takes the finish time value of the last task issued on p_j. A valid free slot can occur only when $T_{pred}(t,j) < T_{avail}(j)$ (line 11). In that case, we search for possible slots (if any) using the insertion scheduling policy and $w_{t,i,1} = min\_w_{t,i}$. If a slot is found then i is inserted in the insertion list (I) and the St variable gets the start time of the free slot. It is important to note that the slots found in lines 2-22 in Algorithm 4, are likely slots not certain. Thus, in the case that a slot has been found, the $EFT(t,j)$ is not given by a single variable but from an inequality, i.e., between [$min\_EFT(t,j)$, $max\_EFT(t,j)$] (lines 18,19). If a slot has not been found, $EFT(t,j)$ value is given by a single value as the minimum and maximum values are the same. The above procedure is applied for all the processors. In line 25 (Algorithm 4), we apply a procedure similar to that in Lines 9-25 (Algorithm 3) in order to reduce the number of candidate processors. However, in this case, the $S(i)$ values are given by an inequality [$S'(i,1)$, $S'(i,2)$]. So, in line 12 (Algorithm 3), $S(i)$ and $S(j)$ are replaced by $S'(i,2)$ and $S'(j,1)$, respectively. Moreover, in line 21 (Algorithm 3), the $S(i)$ is replaced by $S'(i,2)$.

The slots found in lines 2-22 (Algorithm 4), are likely slots not certain. Therefore, no task is simulated (in order to get its $w_{t,i,1}$ value) before we ascertain that its slot is true. This process is held in lines 27-30; the task is simulated only to the processor groups where a) they belong to the simulation list and b) a likely slot has been found. After we get the $w_{t,i,1}$ value of task t, we compute the $EFT(t,j)$ values for all the p_j in group i; this step is applied because in the case that a slot is not true, or the true St value is larger,

Algorithm 4 TSRS with insertion scheduling policy

1: **[$w_{t,i,1}$(), SL()] = TSRS (t) {**
2: **for** $(i = 1, Proc.groups)$ **do**
3: **for** $(j = 1, common.procs)$ **do**
4: **if** $(i \leq p_{ref})$ **then**
5: $min\_w_{t,i} = w_{t,p_{ref},1}$
6: **else**
7: $min\_w_{t,i} = w_{t,p_{ref},1}/50$
8: **end if**
9: Find $T_{pred}(t, j)$
10: $T2 = T_{avail}(j)$, $St = T_{avail}(j)$
11: **if** $(T_{pred}(t, j) < T_{avail}(j))$ **then**
12: Search for possible slots (if any) by using insertion scheduling policy and $w_{t,i,1} = min\_w_{t,i}$
13: **if** (there is a slot) **then**
14: St=start time of the slot
15: put i in insertion scheduling policy list (I)
16: **end if**
17: **end if**
18: $min\_EFT(t, j) = w_{t,p_{ref},1} + max(T_{pred}(t, j), St)$
19: $max\_EFT(t, j) = w_{t,p_{ref},1} + max(T_{pred}(t, j), T2)$
20: Put the min $min\_EFT(t, j)$ value from every group i in $S'(i, 1)$. Put in $S'(i, 2)$, the $max\_EFT(t, j)$ value of $S'(i, 1)$.
21: **end for**
22: **end for**
23:
24: //Reduce the search space
25: Update SL (apply step3 of Algorithm 3, but use $S'(i, 1)$ and $S'(i, 2)$ instead of $S(i)$)
26:
27: **for** $(i = Proc.groups, 1, -1)$ **do**
28: **if** $((i \in I)\&(i \in SL))$ **then**
29: Simulate t on i - get $w_{t,i,1}$
30: Compute the best EFT value (insertion policy enabled) amongst all p_j in group i
31: Reduce the search space - update the SL (apply step2 and step3 of Algorithm 3, but use $S'(i, 1)$ and $S'(i, 2)$ instead of $S(i)$)
32: **end if**
33: **end for**
34: Reduce the search space - update the simulation list (apply step2 and step3 of Algorithm 3)
35: Get the $w_{t,i,1}$ values that $i \in SL$ (if any)
36: Return $w_{t,i,1}$(), SL() }

another processor of the same group may give a better $EFT(t, i)$ value for t. After computing the $EFT(t, i)$ value for a group of processors, we update the simulation list (line 31) - the list is likely to be further reduced and so the *if condition* in line 28 becomes true. The loop in line 27 is executed backwards because fast processors are more likely to be used by HEFT and therefore the SL is more likely to be reduced. In line 34 the simulation list is updated again - in line 34 the $EFT(t, i)$ values are single values.

4.1.3. Extensibility of TSRS

TSRS is applicable to several popular task scheduling heuristics such as HCPT [5], HPS [6], PETS [3], CPOP [2] [46] list scheduling algorithms, [12] [47] clustering algorithms, and others. All the above use the minimum EFT value as the heuristic cost function.

In HCPT, the average earliest start time and the average latest start time are computed using mean values; our method is applicable if $w_{t,p_{ref},1}$ values are used instead of the mean values and apply $TSRS$ to reduce the number of candidate processors. HPS can be extended just by using the $TSRS$ routine. PETS can be extended by computing the average computation cost with $w_{t,p_{ref},1}$ instead of the mean values and using $TSRS$ to reduce the candidate processors. Regarding CPOP, first, we compute the upward and downward rank values by using $w_{t,p_{ref},1}$ instead of the mean values, second we set the HCCP as the critical path processor and third we use $TSRS$ routine. Although TSRS is applicable to heuristics using duplication too, such as [46] and [47], they are not preferred because duplication increases the number of simulations performed. TSRS is applicable to MOHEFT [18] too, which is the first multi-objective optimization proposal that extends a list-based heuristic, but the simulation gain is expected to be very low because MOHEFT builds several intermediate workflow schedules in parallel in each step; however, there are still solutions being 'dominated' by others and therefore they are discarded. TSRS is not applicable to Lookahead [17] and PEFT [1] (in its current form).

Although we have not applied TSRS to the above algorithms, we expect that the makespan degradation (if any) would be insignificant. In Subsection 5.3, we show that for HEFT, the makespan is not degraded. In general, TSRS reduces the number of candidate processors without excluding any processor with a minimum EFT value, apart from the case where the optional step3 in Algorithm 3 is applied to further reduce the scheduling time. However, the $rank_u$ values are now computed using the computation costs on p_{ref} and

not the average ones and this slightly affects the task priority list. So, the reason for any potential makespan degradation (if any) is the new priority list. In [45], the rank function of HEFT algorithm is investigated by using the mean, median, worst and best computation costs; it is shown that for random computation costs (not monotonic as in our case) first, different ways of computing $rank_u$ affects HEFT performance and second, the mean computation costs is not the best choice. In addition to [45], in Subsection 5.3 we show that the mean computation costs do not provide better solutions than the p_{ref} ones.

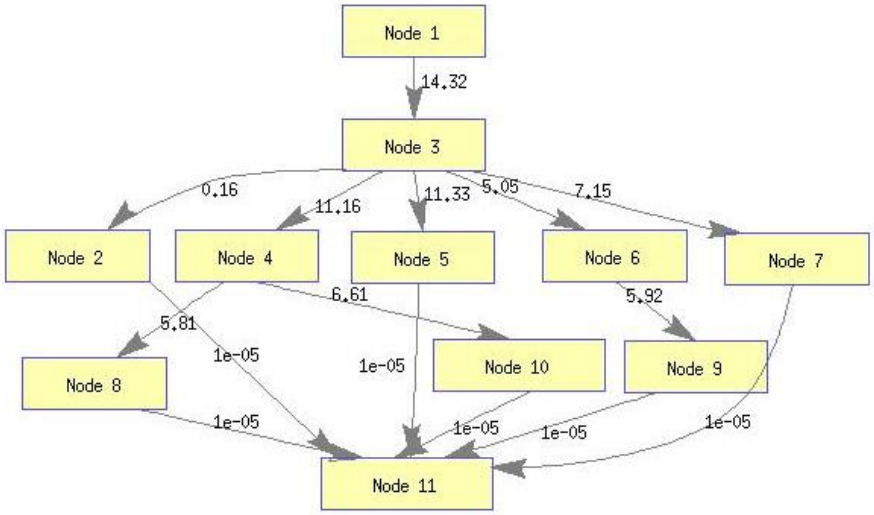
4.2. Multi-Threading Effective Task Scheduling (METS)

In this Subsection, we propose low complexity heuristics to find which tasks are going to be implemented as single-threaded (ST) implementations, which as multi-threaded (MT) implementations, as well as the number of threads used. It is important to note that by scheduling all the tasks as MT implementations, less processing elements are available but with higher CC and vise versa. An example is given in Fig. 1 for two quad-core CPUs. Given that HEFT algorithm assumes rigid (non-moldable) tasks, we have implemented HEFT using ST CPU implementations (SHEFT) only and max-threaded CPU implementations (MHEFT) only. The default HEFT implementation can be considered that of SHEFT. In Fig. 1, we show the schedules that SHEFT and MHEFT give for a sample task graph. Note that t_1 needs 22.60 time units (see table in Fig.1) to run on the one core of P1 and 11.32 time units to run on all the four cores of P1 (MT implementation). Although MHEFT performs better than SHEFT in this example, this is not always the case; a detailed comparison and analysis is performed in Section 5.

METS enhances HEFT algorithm by introducing the following key points:

1. ST implementations are more efficient for tasks with high Communication to Computation Ratio (CCR) values.
2. MT implementations are more efficient when the task parallelism is low.
3. When the task parallelism is high, ST/MT implementations are more efficient when the range of $w_{t,i,j}$ values among different tasks, is low/high, respectively.
4. If a MT task scales well on a multi-core processor, it will scale well to other multi-core processors too, with equal or fewer cores.

METS is given in Algorithm 5. Algorithm 5 enhances HEFT to support moldable tasks. In line 6, the decision whether a task is going to run as ST or



task	$W_{t,1,1}$	$W_{t,1,2}$	$W_{t,1,3}$	$W_{t,1,4}$	$W_{t,2,1}$	$W_{t,2,2}$	$W_{t,2,3}$	$W_{t,2,4}$
1	22.60	17.40	14.15	11.32	21.01	16.18	13.15	10.52
2	17.97	11.65	8.62	6.59	17.06	11.06	8.18	6.25
3	17.30	13.18	10.65	8.49	16.17	12.32	9.95	7.94
4	19.29	13.09	9.91	7.66	17.89	12.14	9.19	7.10
5	32.68	20.02	14.43	10.91	28.72	17.60	12.68	9.59
6	12.48	7.63	5.49	4.15	11.28	6.90	4.97	3.75
7	19.71	14.17	11.07	8.68	17.86	12.85	10.03	7.86
8	32.35	26.60	22.59	18.51	28.54	23.48	19.94	16.33
9	31.04	27.81	25.19	21.43	28.72	25.74	23.31	19.83
10	14.71	12.51	10.88	9.03	13.00	11.05	9.61	7.98
11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

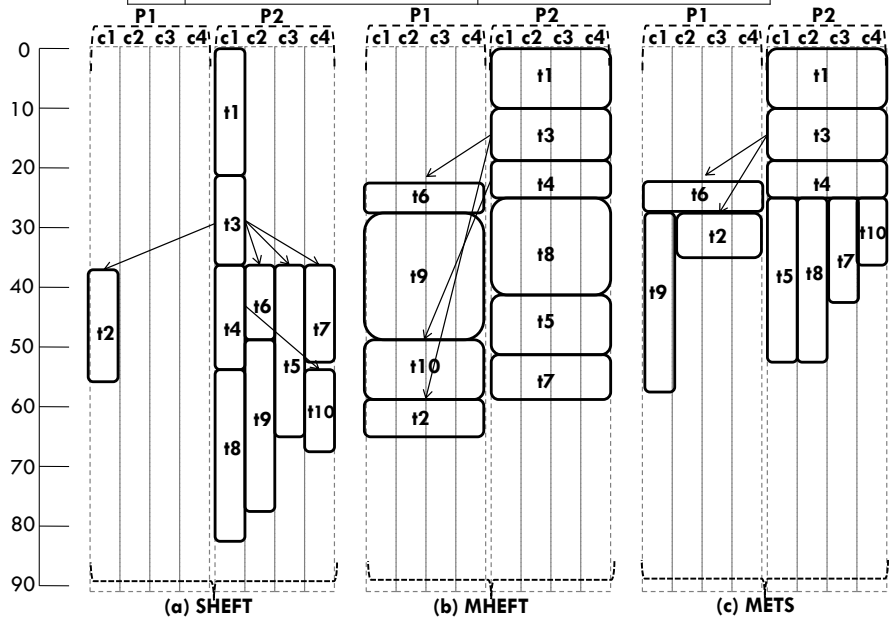


Figure 1: Sample task graph and schedules for a) SHEFT(makespan 83.6), b) MHEFT (makespan 64.7), c) METS (makespan 58.7)

not, is taken. If the task is ST, then HEFT remains unchanged. Otherwise, the implementation giving the minimum EFT for task t is selected, no matter the number of the threads used. All the coefficients in Algorithm 5 are found experimentally. The key points of METS are further explained below.

Regarding the first key point in Algorithm 5, ST implementations are more efficient than the MT ones, for tasks with high CCR values. This is because the data transfer cost from a task to another is minimized when both tasks are executed on the same processor; this means that the data remain in the processor's disk/memory. The more the tasks each processor can handle in parallel (i.e., the more the cores each processor contains), the less the communication cost, as the intra-processor transfer cost is very low. The *if-condition* in line 32 (Algorithm 7) implements the above idea. In line 32, we refer to both parent and child edges for the following reasons. By using a ST implementation for a parent task that gives too much data to its children, we reduce the probability of its children tasks to get data from another processor(s). On the other hand, by using a ST implementation for a child task which gets too much data from its parents, we increase the probability of the other children (with the same parents) to be assigned to the same processor and therefore minimize the data transfer cost.

As far as the second key point is concerned, when the number of the ready tasks is smaller than the number of the processors, there is no reason to save any cores, and thus the implementation giving the minimum EFT value is selected, no matter the number of the cores used (the implementation giving the minimum EFT is not always MT). It is important to note that a) the best MT EFT value is not always the one using the maximum number of threads and b) the MT EFT value is not always lower than the ST EFT, e.g., consider the case where the five out of six cores are not available in the near future. The *if-condition* in line 34 (Algorithm 5) implements the above idea. In Algorithm 5, ST&MT means that we seek for the solution giving the minimum EFT value no matter the number of threads/cores used.

Let us explain the second key point further. Consider there are four identical multi-core processors and only 4 ready tasks. In that case, it is not efficient to save any cores and therefore MT implementations for all the tasks is the best option no matter the number of the threads used. However, if there are 5 ready tasks, it might not be efficient to use MT implementations for all the tasks, because other thread combinations have to be investigated too. This is why we have used the 'Threshold' value in line 30 (Algorithm 5), indicating the number of ready tasks should exist in order to

Algorithm 5 METS

Set the computation costs of tasks and communication costs of edges with mean values
Compute $rank_u$ for all tasks by traversing graph upward, starting from the exit task
Sort tasks in a scheduling list by decreasing order of $rank_u$ values
while there are unscheduled tasks in the list **do**
  Select the first task, t_i, from the list for scheduling
  T=METS_kernel(t_i)
  if T==ST **then**
    //tasks are faced as ST only
    for each processor p_j in the processor-set **do**
      for each CPU core m_k in p_j **do**
        Compute $EFT(t_i, p_j, 1)$ using the insertion-based scheduling policy
      end for
    end for
    Assign task t_i to processor p_j and core m_k that minimizes EFT of task t_i
  else
    //tasks are faced as both ST and MT
    for each processor p_j in the processor-set **do**
      for each thread number f, where $1 \leq f \leq cores$ **do**
        for each CPU core combination, using f threads **do**
          Compute $EFT(t_i, p_j, f)$ using the insertion-based scheduling policy
        end for
      end for
    end for
    Assign task t_i to the CPU cores of p_j that minimize EFT of task t_i
  end if
end while

T=METS_kernel(t) {
A $\leftarrow$ next 6 ready tasks
B $\leftarrow$ next 'Threshold' tasks
C $\leftarrow$ ready tasks that ($Rank_u > 0.7 \times Rank_u(t)$) //tasks executed in near future
if (at least half of the tasks in A contain an edge $c_{n,m}$ (either parent or child edge), where $c_{n,m}/w_{t,p_{ref},1} \geq 1.5$) **then**
  return ST //task is ST only, as the CCR value is high (1st key point)
else if (at least one task in B is not ready) **then**
  //Task parallelism is low
  return ST&MT //this task can be either ST or MT (2nd key point)
else
  // task parallelism is high
  //if the range of $w_{t,p_{ref},1}$ values among different tasks is high (3rd key point)
  if ($Rank_u(t) > (1.3 \times min(Rank_u(C))$) **then**
    if ($factor_{t,p_{ref},f} > good.factor(f)$, f is the max number of threads) **then**
      return ST&MT //this task can be either ST or MT (4th key point)
    else
      if (($factor_{t,p_{ref},\lceil f/2 \rceil} > good.factor(\lceil f/2 \rceil)$) AND (($\lceil f/2 \rceil) > 1$)) **then**
        return ST&MT //this task can be either ST or MT (4th key point)
      else
        return ST //this task is ST only
      end if
    end if
  else
    return ST //this task is ST only
  end if
end if

use ST&MT implementations; in this case, the 'Threshold' value in line 30 must be ($Threshold = 4$). Keep in mind that MT refers to the best multi-threaded solution, no matter how many threads are used. Now consider the case that there are 5 ready tasks and a heterogeneous HW environment with three identical multi-core processors and one GPU (let us assume that the tasks run two times faster on the GPU). One could think that it is not efficient to use MT implementations for all the multi-core processors because one ready task will have to wait until another finishes its execution. However, if the tasks are executed 2 times faster on the GPU than on the CPU (using a max-threaded implementation), the GPU will have executed 2 tasks until the three processors finish their execution. Thus, the GPU 'counts' for 2 CPUs and there is no reason to save any cores. In this case, the 'Threshold' value must be ($Threshold = 5$) and not ($Threshold = 4$). The '$Threshold$' value depends on a) the number of the processors, b) the number of the cores each processor has, c) how faster/slower is one processor to another. The '$Threshold$' value is application independent and depends solely on the HW infrastructure. Thus, it can be found 'off-line'. In Section 4, ($Procs \leq Threshold < 2 \times Procs$), where $Procs$ is the number of the processors.

Regarding the third key point above, i.e., when the number of ready tasks is larger than the '$Threshold$' value, the MT implementations are efficient only in the case where the range of the $w_{t,i,j}$ values for different tasks is high and in particular for the tasks having large $w_{t,i,j}$ values (run the heavy tasks as MT and the light tasks as ST implementations). This is because the core utilization factor value is always lower than the number of the cores and therefore the time needed for a task to be executed as an f-threaded implementation is always higher than executing f tasks. Let us give an example, consider 8 identical tasks ready for execution and two identical 4-core processors. Also consider that the eight tasks need $(10, 6, 4, 3)$ secs to be executed, using $(1, 2, 3, 4)$ threads, respectively. If all the tasks are considered as ST, then 10 secs are required for them to be executed. On the other hand, by using 4-threaded or 2-threaded implementations only, 12 secs are needed. However, if half of the tasks need $(15, 9, 6, 4.5)$ and the other half need $(10, 6, 4, 3)$ seconds to be executed by using $(1, 2, 3, 4)$ threads, respectively, then using only ST implementations is not the best option. If we run the heavy tasks as 4-thread implementations and the light ones as ST ones, then the overall execution time is 14.5 secs, while by using ST only, it is 15 secs. The *if-condition* in line 40 (Algorithm 5) satisfies that only the

tasks with high $w_{t,i,j}$ values are considered as MT. If a task's rank value is larger than 1.3 times the minimum rank value of C (the tasks that are going to be executed in the near future), it is further processed as an ST&MT implementation, otherwise it is assigned as a ST.

In contrast to line 36, where an ST&MT implementation is always selected regardless of whether t is effectively split into multiple threads or not, in line 40, the number of tasks waiting for execution is higher than the number of processors and thus we have to consider the scenario that t may give a low core-utilization factor. Thus, we get $w_{t,p_{ref},f}$ value, where f is the maximum number of threads, and compute the utilization factor. If the factor is large enough, we use a ST&MT implementation, otherwise, we give a second chance for t to be executed with fewer threads, i.e., $\lceil f/2 \rceil$ (line 27). The good utilization factor values used are $(1.6, 2.35, 3.4, 3.9, 4.7)$ for $(2, 3, 4, 5, 6)$ threads, respectively.

Fig. 1 shows an example of the proposed method with Threshold=2. First, tasks 1,3,4,6 are scheduled using 4-threaded implementations (line 36 in Algorithm 5). Tasks 9,5,8,7 are then scheduled using 1-threaded implementations (line 51). Afterwards, task 2 is scheduled using the maximum number of threads (three) because of line 36. Last, task 10 is scheduled according to line 36, but in this case a ST implementation achieves a lower EFT value than a MT.

In terms of time complexity, METS gives $O(e \times p \times cores)$, where cores is the maximum number of physical CPU cores that a multi-core processor supports. The complexity of lines 7-14 is $O(e \times p)$, as in HEFT, where e is the number of edges and p is the overall number of processors (in this context p equals to the number of processors multiplied by the number of their cores). The complexity of lines 16-24 is $O(e \times p \times cores)$. Note that the core combination value in line 19 does not include the exact thread-core mapping, e.g., there is just one combination when $(f == cores)$ and not many. For large graphs and small cores values, the complexity remains $O(e \times p)$.

4.3. TSRS and METS (TSRS+METS)

In this Subsection, METS is applied together with TSRS, in order to optimize for both scheduling time and length. TSRS+METS is shown in Algorithm 7. First, TSRS_for_METS routine finds the candidate processors for task t (line 3). If there is no multi-core candidate processor, the procedure is trivial. Otherwise, the multiple *if-conditions* take place finding whether

the selected processor will use a ST or a ST&MT implementation. In the case that a ST implementation is selected, we simulate t as ST only. Otherwise, if a ST&MT implementation is selected, we simulate t either as ST or MT, not as both.

Algorithm 6 TSRS (Algorithm 3) when it is called by METS (without insertion based scheduling policy)

```
1: [SL(), S(), M()] = TSRS_for_METS (t) {
2:
3: //step1. Compute the EFT values
4: for (i = 1, Proc.groups) do
5:     compute EFT(t, j, 1) for every p_j in group i, by using w_{t,i,1} = w_{t,p_ref,1}
6:     compute EFT(t, j, f) for every p_j in group i and for all the thread combinations f,
       by using w_{t,i,1} = w_{t,p_ref,1} and w_{t,i,f} = w_{t,p_ref,1} × fact.(f)
7:     Put the min EFT(t, j, 1) and EFT(t, j, f) values from every processor group i in
       S(i) and M(i), respectively
8: end for
9:
10: //step2. Reduce the search space
11: Put all the processor groups in the simulation list (SL)
12: for (i = Proc.groups, 2, −1) do
13:     for (j = i − 1, 1, −1) do
14:         if (min(S(i), M(i)) ≤ min(S(j), M(j))) then
15:             remove processor group j from SL
16:         end if
17:     end for
18: end for
19:
20: //step3. this step is optional
21: if (p_ref ∉ HCCP group) then
22:     for (i = 1, Proc.groups − 1) do
23:         if (S(i) ≤ min_EFT_on_p_HCCP) then
24:             remove p_HCCP group from SL
25:         end if
26:     end for
27: end if
28: Return SL(), S(), M() }
```

The new version of TSRS is given in Algorithm 6 and is similar to Algorithm 3. Unlike standalone TSRS, where every processor group has a unique computation cost, in TSRS+METS, a CPU group has as many computation costs as its number of cores (*cores*). However, in TSRS+METS we apply just one simulation for every processor group (line 5-6 in Algorithm 6); we assume

that every task scales equally on different CPUs (Section 2). The $EFT(t, j, 1)$ values for the ST implementations are computed as in Algorithm 3, while the $EFT(t, j, f)$ values for the MT implementations are computed by using median core utilization factor values, $(fact. = 1.5, 2, 2.8, 3, 3.5)$ for $(2, 3, 4, 5, 6)$ cores, respectively (line 5 in Algorithm 6). The best ST and MT EFT value for each processor group is stored into $S(i)$ and $M(i)$, respectively. In step2, a processor group is removed from the SL if both the best ST and MT values are larger than those of another group. Variations with better scheduling lengths but worse scheduling times are feasible by using upper and lower core utilization factors for computing M(i) in line 13. Algorithm 4 (insertion policy) is extended in a similar way, but in this case the makespan improvement is not significant compared to the simulation loss.

In TSRS+METS we use $factor_{t,i,f1}$ value to update $factor_{t,j,f2}$, where $f1 > f2$. We assume that $factor_{t,i,f} = factor_{t,j,f}$, where i, j are different multi-core processors. Moreover, we measure the $factor_{t,i,f1}$ value and update the $factor_{t,j,f2}$ value accordingly, where $f1 > f2$; $factor_{t,j,f2} = (f2 \times factor_{t,i,f1})/f1$. This procedure is applied in lines 24 and 31 (Algorithm 7) in order to update the EFT values on the other processors according to the $factor_{t,p_{ref},f}$ value.

4.3.1. Extensibility of METS and TSRS+METS

TSRS+METS is applicable to all the algorithms that TSRS is applicable to, such as HCPT [5], HPS [6], PETS [3], CPOP [2], [46] [12] [47]. This is straightforward as all the above works use the minimum EFT value as the heuristic cost function. Standalone METS is applicable to all the algorithms that TSRS+METS is applicable to, plus more algorithms such as PEFT [1], lookahead [17] and MOHEFT [18].

Regarding PEFT [1], it uses the Optimistic EFT value ($O_{EFT} = EFT + OCT$) as a heuristic cost function. METS can be applied to PEFT as follows. The OCT values are computed exactly as in [1], i.e., $w_{t,i,1}$ values are used, but in the processor selection phase, METS determines whether a ST or MT implementation is applied and the number of threads used. Regarding Lookahead [17], the appropriate resource is selected that minimizes either the maximum EFT value for the given task's children or the weighted average of EFT value; as in PEFT, our method can decide whether a ST or MT implementation is applied as well as the number of the threads.

Algorithm 7 TSRS+METS

1: $[\mathbf{w_{t,i,thr}}()$, **SL()**$]$ = **TSRS+METS (t) {**
2:
3: **[SL(),S(),M()]=TSRS_for_METS(t);**
4: **if** (SL contains no multi-core processor) **then**
5: Get the $w_{t,i,1}$ values that $i \in SL$ (if any) - $thr = 1$
6: **else**
7: A $\leftarrow$ next 6 ready tasks
8: B $\leftarrow$ next 'Threshold' tasks
9: C $\leftarrow$ ready tasks that $(Rank_u > 0.7 \times Rank_u(t))$ //tasks executed in near future
10: **if** (at least half of the tasks in A contain an edge $c_{n,m}$ (either parent or child edge), where $c_{n,m}/w_{t,p_{ref},1} \geq 1.5$) **then**
11: $[\mathbf{w_{t,i,1}}()$, **SL()**$]$ = **kernel (ST,t);** //processors are faced as ST only
12: **else if** (at least one task in B is not ready) **then**
13: // Task parallelism is low.
14: // Use the implem. giving the min EFT, no matter the # of the threads
15: $[\mathbf{w_{t,i,thr}}()$, **SL()**$]$= **kernel (ST&MT,t);**
16: **else**
17: // task parallelism is high
18: //if the range of $w_{t,p_{ref},1}$ values among diff. tasks is high
19: **if** $(Rank_u(t) > (1.3 \times min(Rank_u(C))$) **then**
20: **Get** $\mathbf{w_{t,p_{ref},f}}$, where f is the max number of threads in SL
21: $factor_{t,p_{ref},f} = w_{t,p_{ref},1}/w_{t,p_{ref},f}$
22: **if** $(factor_{t,p_{ref},f} > good.factor(f))$ **then**
23: //Use the implem. giving the min EFT, no matter the # of threads
24: Use $factor_{t,p_{ref},f}$ to update EFT to other procs
25: $[\mathbf{w_{t,i,thr}}()$, **SL()**$]$ = **kernel (ST&MT,t);**
26: **else**
27: **Get** $\mathbf{w_{t,p_{ref},\lceil f/2 \rceil}}$
28: $factor_{t,p_{ref},\lceil f/2 \rceil} = w_{t,p_{ref},1}/w_{t,p_{ref},\lceil f/2 \rceil}$
29: **if** ($(factor_{t,p_{ref},\lceil f/2 \rceil} > good.factor(\lceil f/2 \rceil))$ AND $((\lceil f/2 \rceil) > 1)$) **then**
30: //Use the implem.giving the min EFT, no matter the # of threads
31: Use $factor_{t,p_{ref},\lceil f/2 \rceil}$ value to update EFT to other processors
32: $[\mathbf{w_{t,i,thr}}()$, **SL()**$]$ = **kernel (ST&MT,t);**
33: **else**
34: $[\mathbf{w_{t,i,1}}()$, **SL()**$]$ = **kernel (ST,t);** //processors are faced as ST only
35: **end if**
36: **end if**
37: **else**
38: $[\mathbf{w_{t,i,1}}()$, **SL()**$]$ = **kernel (ST,t);** //processors are faced as ST only
39: **end if**
40: **end if**
41: **end if**
42: **Return** $w_{t,i,thr}()$, **SL();** **}**
43:
44: $[\mathbf{w_{t,i,thr}}()$, **SL()**$]$ = **kernel (T,t) {**
45: **if** $(T == ST)$ **then**
46: **[SL(),S(),M()] = TSRS_for_METS(t)** - by using S() only, not M()
47: Get the $w_{t,i,1}$ values that $i \in SL$ (if any) - $thr = 1$
48: **else**
49: **[SL(),S(),M()] = TSRS_for_METS(t)**
50: Get the $w_{t,i,thr}$ values (if any) where $i \in SL$ and thr is the number of threads of the $\min(S(i), M(i))$
51: **end if**
52: **Return** $\mathbf{w_{t,i,thr}}()$, **SL(); }**

5. Experimental Results

This section shows the application of TSRS and METS to HEFT algorithm. We have evaluated our work to 14580 different random DAGs and 5 real world applications.

The comparison metric used for evaluating the schedule's length is speedup (Eq. 6) which is computed by dividing the sequential execution time by the parallel execution time (makespan). The sequential execution time is computed by assigning all tasks to a single processor that minimizes the cumulative of the computation costs, i.e., Highest Computational Capability Processor (HCCP); if the HCCP is a multi-core processor, then the numerator of Eq. 6 refers to max-threaded implementations.

$$Speedup = \frac{min_{p_j \in P}\{\sum_{t_i \in V} w_{i,j,f}\}}{makespan} \tag{6}$$

The simulation gain (Eq. 7) is given by dividing the overall number of simulations needed to generate matrix W by the number of simulations performed by our method (simulations). The numerator is given by $((\sum_{i=1}^{P} c_i + co) \times tasks)$, where P is the number of multi-core processor groups, c_i is the number of group i cores and co is the number of non-CPU groups.

$$Simulation.gain = \frac{(\sum_{i=1}^{P} c_i + co) \times tasks}{simulations} \tag{7}$$

5.1. Hardware Infrastructure

The Hardware (HW) infrastructure used in this paper, consists of 9 different groups of processors (6 multi-core CPU and 3 GPU groups), 3 common processors in each group (27 processors in total) and 6 cores per processor at maximum. The groups of processors are sorted in increasing computational capability (CC), i.e., $(w_{t,9,1} \leq w_{t,8,1} \leq ... \leq w_{t,1,1})$. The HW infrastructure is described by $D.P(9)$, $C.P(3)$ and $cores(6)$ arrays, giving the number of different processors, common processors and cores, respectively. So, for instance, the HW infrastructure described by $\{D.P(0,0,0,1,1,1,1,0,0)$, $C.P(0,0,0,1,2,3,1,0,0)$ and $cores(0,0,0,2,4,6)\}$, refers to one 2-core CPU of type4, two 4-core CPUs of type5, three 6-core CPUs of type6 and one GPU of type7. The GPUs are of higher CC than CPUs and therefore they always refer to processors with number 7, 8 and 9. Moreover, to make the HW infrastructure more realistic, we assume that $(w_{t,7,1} \leq 5 \times w_{t,6,1})$.

5.2. Random graphs

First, we evaluate our work to random generated application graphs with random computation/communication costs. For this purpose, we used the synthetic DAG generation program Daggen [48] with five different parameters defining the DAG's shape:

- n: number of DAG nodes. Four different values are used $n = (50, 100, 200, 300)$.
- fat: this parameter affects the height and the width of 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 with low task parallelism, whereas a large value induces a fat DAG with a high degree of parallelism. The following fat values are used $fat = (0.2, 0.5, 0.8)$.
- 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, $density = (0.2, 0.5, 0.8)$.
- 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, $regularity = (0.2, 0.5, 0.8)$.
- jump: indicates that an edge can go from level l to level $l + jump$, $jump = (1, 2, 4)$.

To obtain the random computation and communication costs, the following parameters have been used:

- β_w (Range percentage of computation costs among different tasks for p_{ref}): A high value implies wider computation costs among tasks while a low value implies narrower costs. $\beta_w = (0.5, 1, 1.5)$. In Eq. 8, $\overline{w}$ is the average computation cost of the DAG and is selected randomly.

$$\overline{w} \times (1 - \frac{\beta_w}{2}) \leq w_{t,p_{ref},1} \leq \overline{w} \times (1 + \frac{\beta_w}{2}) \tag{8}$$

- CCR: Communication-to-Computation Ratio: ratio of the sum of the edge weights to the sum of the node weights on p_{ref}, CCR=(0.1, 0.2, 0.5, 1, 2, 5, 10).

- β_c (Range percentage of communication costs among the edges of the DAG): A high value implies wider communication costs among different edges while a low value implies narrower costs. β_c is given by the following formula where $\overline{c}$ is the average communication c value of the DAG and $\overline{c} = \overline{w} * CCR$. $\beta_c = (0.5, 1, 1.5)$.

$$\overline{c} \times (1 - \frac{\beta_c}{2}) \leq c_{i,j} \leq \overline{c} \times (1 + \frac{\beta_c}{2}) \quad (9)$$

All the above generate 14580 different DAGs.

The computation costs for the other processors are generated according to the computation costs on p_{ref}. The computation costs of the remaining processors (p_i) are random values within the following range: $w_{t,p_{ref},1} \times R(i,1) \leq w_{t,i,1} \leq w_{t,p_{ref},1} \times R(i,2)$, where R=(2,2.5; 1.8,2; 1.4,1.5; 1.2,1.3; 1.05,1.15; 1,1; 0.12,0.2; 0.08,0.18; 0.05,0.15). Regarding multi-threaded computation costs, we have used random realistic speedup range values, i.e., $w_{t,i,f} = w_{t,i,1} \times speedup(f)$, where the speedup value is a random value within the following range $(1.1, 1.9)$, $(1.2, 2.8)$, $(1.3, 3.7)$, $(1.4, 4.5)$,$(1.5, 5.4)$, for $(2, 3, 4, 5, 6)$ threads, respectively. We assume that $(w_{t,i,f1} \leq w_{t,i,f2})$, where $f1 \geq f2$. It is important to note that we have used both wider and narrower values than R and the results are similar.

5.3. Evaluation of TSRS

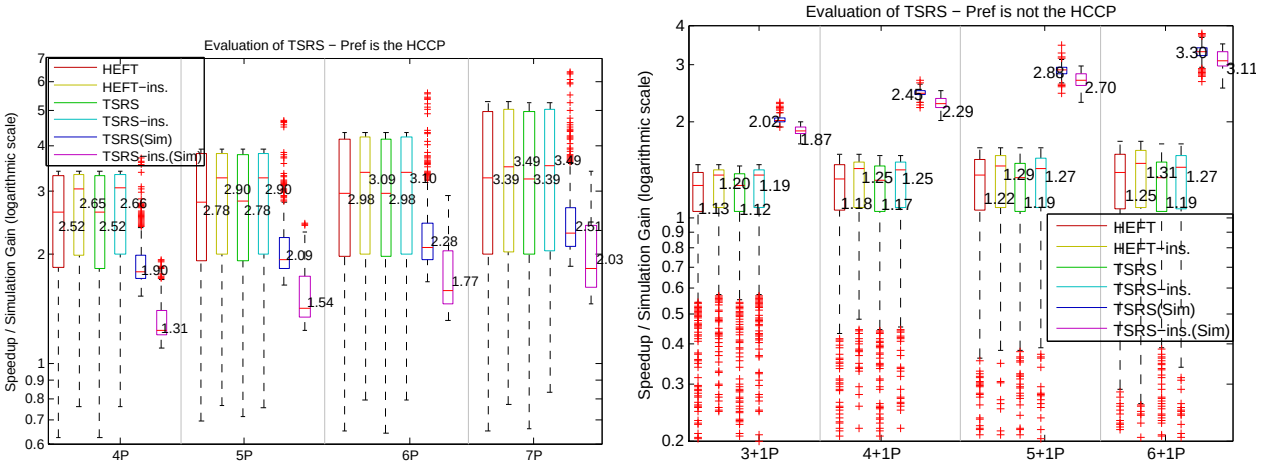


Figure 2: Evaluation of TSRS (972 different DAGs)

In this Subsection, TSRS is evaluated. The results are illustrated by using boxplots in Matlab. On each box, the central red line indicates the

median value, the displayed value shows the mean, and the bottom and top edges of the box indicate the 25th and 75th percentiles, respectively. The whiskers extend to the most extreme data points not considered outliers, and the outliers are plotted individually using the '+' symbol.

The TSRS is evaluated in the case where p_{ref} is a HCCP and not (Fig. 2). In the second case, the extra and optional loop kernel is executed (step3 in Algorithm 3), slightly degrading HEFT's scheduling length but increasing the simulation gain. The 'Sim' indicates the simulation gain, while the 'ins.' indicates that the insertion scheduling policy is used. In this subsection, all the processors are either single-core CPUs or GPUs. The top figure in Fig. 2, refers to single-core CPUs only where '4P' and '5P' indicate 4 and 5 different single-core CPUs, respectively. The bottom figure in Fig. 2 refers to single-core CPUs and one GPU, where the '3+1P' value indicates 3 different single-core CPUs and one GPU. In Fig. 2, 972 different DAGs have been used (all different fat, regularity, density and jump combinations) with $n = 100$,$CCR = (0.1, 0.5, 2, 10)$, $\beta_w = \beta_c = (0.5, 1, 1.5)$ as well as several processor configurations. In the first figure of Fig. 2, TSRS uses a HCCP as p_{ref}, while in the second it uses a 2nd HCCP, and therefore, the $Rank_u$ values have been computed by using a 1st/2nd HCCP, respectively. It is important to note that the speedup values are low in the second figure, because the numerator of Eq. 6 refers to a fast GPU ($(w_{t,7,1} \leq 5 \times w_{t,6,1})$).

As far as the quality of the output schedule length is concerned (makespan), when p_{ref} is a HCCP and therefore step3 of Algorithm 3 is not used, the TSRS makespan is approximately the same as that of the standalone HEFT, in all cases. On the other hand, when p_{ref} is not a HCCP, the TSRS makespan is slightly degraded (Subsection 4.1); the more the processor groups are, the more the makespan degradation, as the '$min\_EFT\_on\_p_{HCCP}$' in Algorithm 3 is compared with more processor groups. Furthermore, both methods perform better by using the insertion scheduling policy but the gains are small.

Regarding the simulation gain values, when p_{ref} is not a HCCP, the gain values are larger on average, but when p_{ref} is a HCCP, the gain values are very wide giving both larger and smaller values. This is because in the latter case the number of candidate processors is reduced by using just one phase/step (step2 in Algorithm 3), while in the first case two steps are used (step2 and step3). This makes the latter case more sensitive to different DAGs and therefore giving both very high and low gain values. As it was expected, by using the insertion scheduling policy lower simulation gain values occur because in Algorithm 4 the number of computation costs needed is higher.

It is important to note, that the purpose of this subsection is not to compare the gain values between the first and second figure of Fig. 2 as a) they both refer to different hardware configurations, b) the gain values strongly depend on the DAG set used.

5.4. Evaluation of METS

In this Subsection, standalone METS is evaluated (Fig. 3). HEFT algorithm assumes rigid (non-moldable) tasks and therefore for a comparison to be made, we have implemented HEFT to use either ST CPU implementations (SHEFT) only or max-threaded CPU implementations (MHEFT) only. Note that the default/original version of HEFT is SHEFT. Regarding our method, $p_{ref} = 6$ in all cases; in METS the p_{ref} is always the CPU with the maximum number of cores.

Among all the DAG parameters given in Subsection 5.2, METS is affected the most by the type of the processors (multi-core CPUs or non-CPUs), the ratio of the number of tasks to the number of processors and the tasks parallelism (fat value). Skinny DAGs give lower speedup values (in all methods), compared to the fat DAGs. This is because in skinny DAGs the task parallelism is low and task dependencies force many processors to remain idle, while in fat DAGs, the task parallelism is high and most of the processors work in parallel, further increasing the speedup value. SHEFT is more efficient than MHEFT when the task parallelism is medium/high, as by providing more CPU cores, more tasks are executed in parallel. On the other hand, when the task parallelism is low, MHEFT gives higher speedup values, as it is preferable to use fewer processors but with higher CC.

In Fig. 3, METS is evaluated for all the parameter combinations in Subsection 5.2 (14580 DAGs) and six different HW configurations. The left three HW configurations in Fig. 3 refer to HW platforms where only multi-core CPUs are used, while the right three configurations refer to platforms with both CPUs and GPUs. When only CPUs are used, the heuristics given in Subsection 4.3 perform very well and give high speedup values. In the first HW configuration, where the number of the available processors is small, SHEFT performs much better than MHEFT, while on the last is exactly the opposite; by increasing the number of the processors, MHEFT outperforms SHEFT, as it uses processors with higher CC. Our method provides better makespan values in all cases.

Regarding the right three HW configurations in Fig. 3, METS gives significant speedup values, but lower speedup values compared to the left three

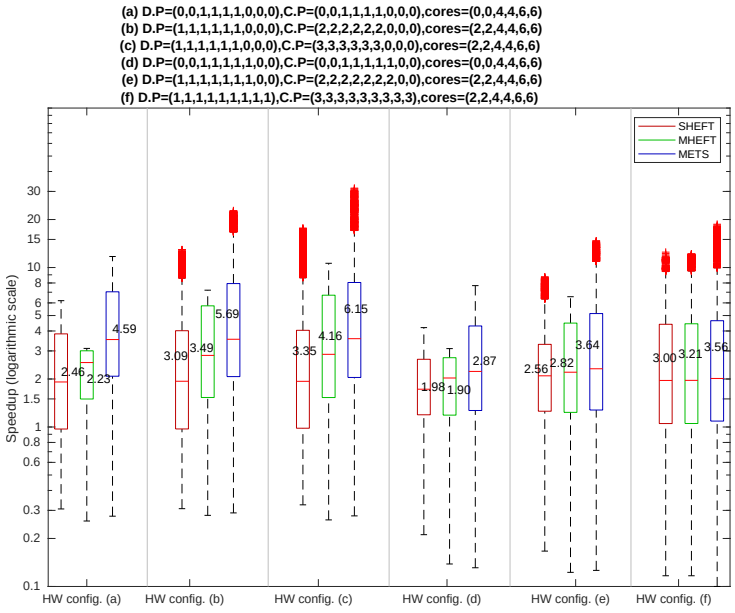


Figure 3: Evaluation of METS (14580 different DAGs)

HW configurations. The reason lies in the fact that HEFT is a greedy algorithm as it always chooses the processor giving the minimum EFT value; therefore, if the GPUs are many times faster than the CPUs, they never become idle and push aside the CPUs; thus, most of the tasks are scheduled on the GPUs, and the key points of METS have a lower impact. In the last configuration, the proposed method gives low speedup values because the number of processing units per task is high. Note that when the number of processing units per task is high, the TS problem has a lower impact compared to the case where the number of processing units per task is low.

5.5. Evaluation of TSRS+METS

In this Subsection, TSRS+METS is evaluated (Fig. 4). We have evaluated TSRS+METS without using the insertion scheduling policy because the makespan improvement is insignificant compared to the simulation loss. Unlike our method, SHEFT and MHEFT use the insertion scheduling policy. $p_{ref} = 6$ in all cases.

As it can be observed, METS provides better makespan values than TSRS+METS, because a) in Algorithm 6 the EFT values are not computed

by using the real computation costs, b) TSRS+METS does not use the insertion policy.

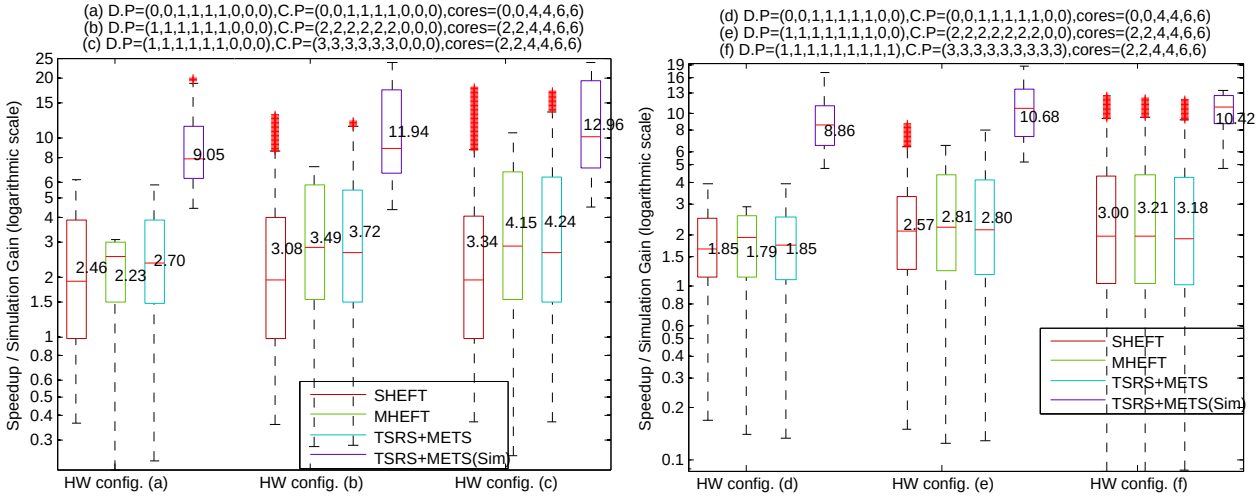


Figure 4: Evaluation of TSRS+METS (14580 different DAGs)

As in METS, among all the DAG parameters given in Subsection 5.2, the studied methods are affected the most by the type of the processors (multi-core CPUs or non-CPUs), the ratio of the number of tasks to the number of processors and the tasks parallelism (fat value). Skinny DAGs give lower speedup but higher simulation gain values, while fat DAGs give higher speedup but lower simulation gain values. As far as the simulation gain values are concerned, in skinny DAGs, most of the tasks are assigned to the high CC processors while the lower CC ones remain idle a significant amount of time; in that case $T_{Avail}(i) < T_{pred}(t,i)$ more often and therefore $EST(t,i) = max(T_{Avail}(i), T_{pred}(t,i)) = T_{pred}(t,i)$; thus, Algorithm 3 is more likely to reduce the number of simulations as $T_{pred}(t,i)$ value remains constant among different processors, e.g., most of the constant values in Eq.4 are equal.

In Fig. 4, TSRS+METS is evaluated for all the parameter combinations in Subsection 5.2 (14580 DAGs) and six different HW configurations. The left figure in Fig. 4 refers to HW platforms where only multi-core CPUs are used, while the right refers to both CPUs and GPUs. When only CPUs are used, the heuristics given in Subsection 4.3 perform very well and give significant speedup values. In the first HW configuration, where the number of the available processors is small, SHEFT performs much better than MHEFT, while on the last is exactly the opposite; by increasing the number of the

processors, MHEFT outperforms SHEFT, as it uses processors with higher CC. Our method provides better makespan values in all cases. Moreover, the higher the number of the processors, the higher the simulation gain as the lower CC processors are more likely to be excluded from the SL.

Regarding the right figure in Fig. 4, when very fast GPUs are used too, TSRS+METS gives improved but low makespan gains over the default HEFT algorithm (SHEFT) in most cases. Furthermore, TSRS+METS gives similar schedule lengths to the best of SHEFT and MHEFT, i.e., similar makespan values to SHEFT/MHEFT when the number of processors is small/high, respectively. The higher the number of the GPUs, the less the makespan gain of TSRS+METS. The reason lies in the fact that HEFT is a greedy algorithm as it always chooses the processor giving the minimum EFT value; therefore, if the GPUs are many times faster than the CPUs, they never become idle and push aside the CPUs; thus, most of the tasks are scheduled on the GPUs, and the key points of METS have lower impact. This is why all three methods give close makespan values in cases (d) and (f), where the number of GPUs is high compared to the CPUs. In the last case (f), where there are nine GPUs, most of the tasks are scheduled on the GPUs. Still, the proposed TSRS+METS follows the trend of the best of the two. Note that the default HEFT algorithm is SHEFT not MHEFT. As far as the simulation gain is concerned, it is higher when no GPU exists. In this case, p_{ref} and not a GPU is the fastest and the most preferable processor and thus most of the tasks are scheduled on p_{ref} whose computation costs have already been computed in the initialization phase. On the other hand, when GPUs exist, most of the tasks are allocated on the GPUs ($p_{ref} > 6$) while $p_{ref} = 6$ and as a consequence a larger number of extra simulations is required.

5.6. Evaluation on Real World Applications

In addition to the random graphs, we evaluated our work to 5 real world applications (Fig. 5). These are Montage, CyberShake, Epigenomics, LIGO, SIPHT [49] [50]. We have used small, medium and large graphs for each one of the 5 applications (from 50 up to 200 tasks, Fig. 5) as well as real communication and computation costs for $w_{t,p_{ref},1}$, taken from [49] [50]. The computation costs for the other processors and the multi-threaded implementations, have been selected as random values as in Subsection 5.2.

METS gives impressive speedup values in all cases (Fig. 5). TSRS+METS gives improved makespan values over the default HEFT algorithm (SHEFT) in most cases. Furthermore, TSRS+METS follows the trend of the best between SHEFT and MHEFT in all cases, and it is even better than both

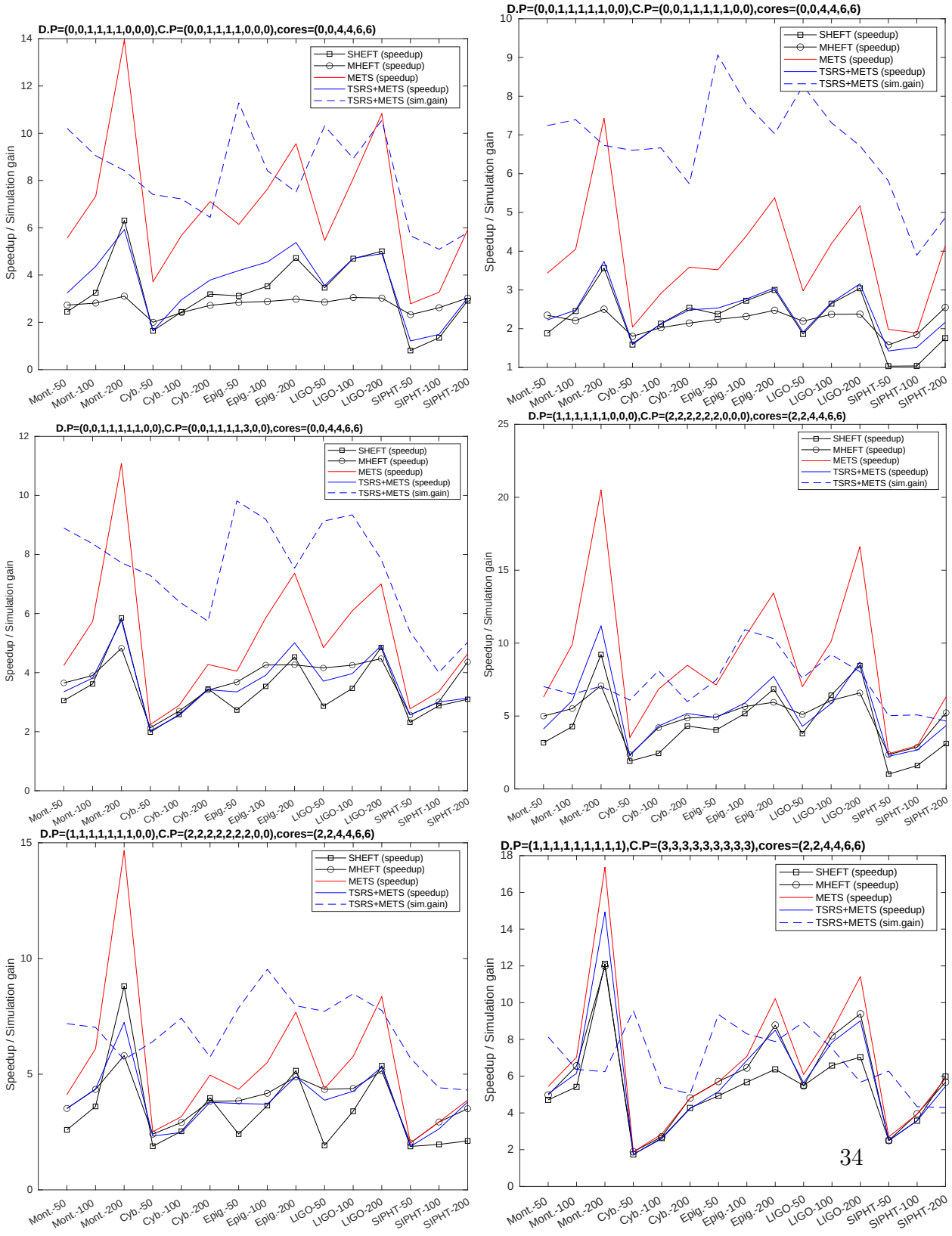


Figure 5: Evaluation of METS and TSRS+METS, for 5 real world applications. Six different hardware configurations are shown. The default version of HEFT is SHEFT.

when no GPUs are used. TSRS+METS achieves simulation gain values from $x4.2$ up to $x11.6$.

In Montage application most of the time is spent in I/O operations and therefore SHEFT performs better than MHEFT in most cases, especially when the number of tasks is high. By using more ST implementations, the data transfer cost is reduced, as tasks are more likely to be executed on the same processor. The more the tasks each processor handles in parallel, the less the communication cost as the intra-processor transfer cost is very low. Regarding CyberShake, the communication costs are higher than the computation costs too, but not as much as in Montage; in CyberShake, by proving more cores, SHEFT and MHEFT give close speedup values. When the number of the processors is low, METS achieves up to $x2.3/x2.2$ times better schedule lengths for Montage/CyberShake, respectively. Note that when the number of processors is low, selecting the right processor for each task highly impacts the makespan value. Our methods perform better mainly because of the first key point in Subsection 4.2.

Regarding Epigenomics and LIGO, the communication costs are comparable to the computation costs. SHEFT performs better than MHEFT when the number of processors is low (less or equal to 5), while MHEFT performs better than SHEFT when the number of processors is higher. This is because in the first case, MHEFT uses only a few processing elements, e.g., in the top left figure in Fig. 5, SHEFT uses 20 processing elements (CPU cores) of low CC while MHEFT uses just 4 processing elements but of higher CC. As in Montage and CyberShake, METS achieves high speedup values when the ratio of processors per task is low, and lower speedup values when the ratio is high. This is more clear in the last figure where METS is way more efficient for large DAGs.

SIPHT is primarily a CPU-bound workflow and most of its runtime is spent on a few tasks only. This is why MHEFT performs always better than SHEFT. Both METS and TSRS+METS give lower speedup values in SIPHT, compared to the other applications. This is because most of the execution time is spent on a few tasks only and if one of these tasks is assigned to a low CC processor, then the overall makespan is degraded.

6. Conclusions and Future Work

We have presented two methods for effective TS in HCS. Although we have integrated both methods to HEFT algorithm only, we have shown that both methods are applicable to other algorithms too.

TSRS modifies HEFT's processor selection phase in order to discard all the processors which cannot minimize the heuristic cost function, regardless of their computation costs; this way, the DAG computation costs required by HEFT become limited. Although TSRS, never excludes a processor minimizing the heuristic cost function (here EFT), it slightly affects the task priority list. However, the experimental results show that for monotonic computation costs the output makespan is not degraded; as in [45], we show that the mean $Rank_u$ computation costs is not the best choice. The insertion scheduling policy is not preferred as the makespan improvement is not significant comparing to the simulation loss.

METS refers to low-complexity heuristics to find which tasks are going to be split into multiple threads as well as the number of the threads used. By enhancing HEFT with METS, high makespan gain values are achieved over the default HEFT algorithm (SHEFT) in all cases (from x1.1 up to x2.3). For large graphs and CPUs with not many cores, HEFT's complexity remains unchanged.

TSRS and METS are also applied together to optimize for both scheduling time and length. We have shown that by enhancing HEFT with TSRS+METS, significant/low speedup values are achieved over the default HEFT algorithm (SHEFT) in HCS without/with fast coprocessors (average gain x1.12); this is because HEFT is a greedy algorithm and it always selects the processor giving the minimum EFT value; therefore, the fast coprocessors never become idle and push aside the slower CPUs, meaning that most of the tasks are executed on the coprocessors. In this case, the key points of METS (Subsection 4.2) have a lower impact. Standalone METS provides better makespan values than TSRS+METS in all cases, as first the computation cost matrix is known and second it includes the insertion-based scheduling policy.

In our future work, we intend to apply and evaluate TSRS, METS and TSRS+METS to other TS algorithms such as HCPT, HPS, PETS, CPOP and others, in terms of makespan and simulation gain.

Acknowledgments

This work is partly supported by the European Commission under H2020-ICT-20152 contract 687584 - Transparent heterogeneous hardware Architecture deployment for eNergy Gain in Operation (TANGO) project.

References

References

[1] H. Arabnejad, J. G. Barbosa, List scheduling algorithm for heterogeneous systems by an optimistic cost table, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2014) 682–694. doi:10.1109/TPDS.2013.57.
URL http://dx.doi.org/10.1109/TPDS.2013.57

[2] H. Topcuouglu, S. Hariri, M.-y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274. doi:10.1109/71.993206.
URL http://dx.doi.org/10.1109/71.993206

[3] E. Ilavarasan, P. Thambidurai, Low complexity performance effective task scheduling algorithm for heterogeneous computing environments, Journal of Computer Sciences 3 (2007) 94–103.

[4] O. Beaumont, V. Boudet, F. Rastello, Y. Robert, Matrix multiplication on heterogeneous platforms, IEEE Transactions on Parallel and Distributed Systems 12 (10) (2001) 1033–1051.

[5] T. Hagras, J. Janecek, A simple scheduling heuristic for heterogeneous computing environments, in: 2nd International Conference on Parallel and Distributed Computing, ISPDC'03, Washington, DC, USA, 2003, pp. 104–110.
URL http://dl.acm.org/citation.cfm?id=1899290.1899305

[6] E. Ilavarasan, P. Thambidurai, R. Mahilmannan, High performance task scheduling algorithm for heterogeneous computing system, in: Proceedings of the 6th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 193–203.

[7] V. Kelefouras, K. Djemame, Workflow simulation aware and multithreading effective task scheduling for heterogeneous computing, in: 2018 IEEE 25th International Conference on High Performance Computing (HiPC), 2018, pp. 215–224.

[8] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, D. Trystram, Scheduling Independent Moldable Tasks on Multi-Cores

with GPUs, IEEE Transactions on Parallel and Distributed Systems (2017) 14doi:10.1109/TPDS.2017.2675891.

[9] S. Hunold, One step towards bridging the gap between theory and practice in moldable task scheduling with precedence constraints, Concurrency and Computation: Practice and Experience 27 (4) (2015) 1010–1026. doi:10.1002/cpe.3372.
URL http://dx.doi.org/10.1002/cpe.3372

[10] M. I. Daoud, N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 68 (4) (2008) 399 – 409. doi:https://doi.org/10.1016/j.jpdc.2007.05.015.
URL http://www.sciencedirect.com/science/article/pii/S0743731507000834

[11] A. K. Singh, M. Shafique, A. Kumar, J. Henkel, Mapping on multi/many-core systems: Survey of current and emerging trends, in: Proceedings of the 50th Annual Design Automation Conference, DAC '13, ACM, New York, NY, USA, 2013, pp. 1:1–1:10. doi:10.1145/2463209.2488734.
URL http://doi.acm.org/10.1145/2463209.2488734

[12] C. Boeres, J. V. Filho, V. E. F. Rebello, A cluster-based strategy for scheduling task on heterogeneous processors, in: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 214–221. doi:10.1109/SBAC-PAD.2004.1.
URL https://doi.org/10.1109/SBAC-PAD.2004.1

[13] S. Ranaweera, P. Agrawal, A task duplication based scheduling algorithm for heterogeneous systems, in: Proceedings of the 14th International Symposium on Parallel and Distributed Processing, IPDPS '00, IEEE Computer Society, Washington, DC, USA, 2000, pp. 445–.
URL http://dl.acm.org/citation.cfm?id=846234.849341

[14] G. Theodoridis, N. Vassiliadis, S. Nikolaidis, An integer linear programming model for mapping applications on hybrid systems, IET Computers & Digital Techniques 3 (2009) 33–42.

[15] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, F. Charot, Constraint programming approach to reconfigurable processor extension generation and application compilation, ACM Trans. Reconfigurable Technol. Syst. 5 (2) (2012) 10:1–10:38. doi:10.1145/2209285.2209289.
URL http://doi.acm.org/10.1145/2209285.2209289

[16] A. Emeretlis, G. Theodoridis, P. Alefragis, N. S. Voros, A hybrid ILP-CP model for mapping directed acyclic task graphs to multicore architectures, in: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014, 2014, pp. 176–182. doi:10.1109/IPDPSW.2014.24.
URL https://doi.org/10.1109/IPDPSW.2014.24

[17] L. F. Bittencourt, R. Sakellariou, E. R. M. Madeira, Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm, in: 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 27–34. doi:10.1109/PDP.2010.56.
URL http://dx.doi.org/10.1109/PDP.2010.56

[18] J. J. Durillo, R. Prodan, Multi-objective workflow scheduling in amazon EC2, Cluster Computing 17 (2) (2014) 169–189. doi:10.1007/s10586-013-0325-0.
URL https://doi.org/10.1007/s10586-013-0325-0

[19] E. U. Munir, S. Mohsin, A. Hussain, M. W. Nisar, S. Ali, Sdbats: A novel algorithm for task scheduling in heterogeneous computing systems., in: IPDPS Workshops, IEEE, 2013, pp. 43–53.
URL http://dblp.uni-trier.de/db/conf/ipps/ipdps2013w.html

[20] S. Sandokji, F. E. Eassa, Dynamic variant rank heft task scheduling algorithm toward exascle computing, Procedia Computer Science.

[21] H. Mahmoud, M. Thabet, M. Khafagy, F. Omara, An efficient load balancing technique for task scheduling in heterogeneous cloud environment, Cluster Computing 24. doi:10.1007/s10586-021-03334-z.

[22] T. McSweeney, N. S. Walton, M. Zounon, An efficient new static scheduling heuristic for accelerated architectures, Computational Science – ICCS 2020 12137 (2020) 3 – 16.

[23] H. R. Faragardi, M. R. Saleh Sedghpour, S. Fazliahmadi, T. Fahringer, N. Rasouli, Grp-heft: A budget-constrained resource provisioning scheme for workflow scheduling in iaas clouds, IEEE Transactions on Parallel and Distributed Systems 31 (6) (2020) 1239–1254.

[24] X. Zhou, G. Zhang, J. Sun, J. Zhou, T. Wei, S. Hu, Minimizing cost and makespan for workflow scheduling in cloud using fuzzy dominance sort based heft, Future Generation Computer Systems 93 (2019) 278 – 289.

[25] S. Muhammad, H. Zahid, L. Mustapha, W. Muhammad, T. Shanshan, An evolutionary computing-based efficient hybrid task scheduling approach for heterogeneous computing environment, J. Grid Comput. 19 (1) (2021) 11.

[26] M. Sulaiman, Z. Halim, M. M. Waqas, D. Aydın, A hybrid list-based task scheduling scheme for heterogeneous computing, J. Supercomput. 77 (2021) 10252–10288.

[27] A. M. Chirkin, A. S. Z. Belloum, S. V. Kovalchuk, M. X. Makkes, Execution time estimation for workflow scheduling, in: 9th Workshop on Workflows in Support of Large-Scale Science, WORKS '14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 1–10. doi:10.1109/WORKS.2014.11.
URL http://dx.doi.org/10.1109/WORKS.2014.11

[28] V. Korkhov, Hierarchical resource management in grid computing, Ph.D. thesis, University of Amsterdam (2009).

[29] A. Afzal, J. Darlington, A. S. McGough, Stochastic workflow scheduling with qos guarantees in grid computing environments, in: 5th International Conference on Grid and Cooperative Computing GCC, Changsha, Hunan, China, 21-23 October 2006, Proceedings, 2006, pp. 185–194. doi:10.1109/GCC.2006.89.
URL https://doi.org/10.1109/GCC.2006.89

[30] P. Dutot, H. Casanova, F. Suter, T. N'Takpé, Scheduling parallel task graphs on (almost) homogeneous multicluster platforms, IEEE Transactions on Parallel & Distributed Systems 20 (2009) 940–952. doi:10.1109/TPDS.2009.11.
URL doi.ieeecomputersociety.org/10.1109/TPDS.2009.11

[31] L. Fan, F. Zhang, G. Wang, Z. Liu, An effective approximation algorithm for the malleable parallel task scheduling problem, J. Parallel Distrib. Comput. 72 (5) (2012) 693–704. doi:10.1016/j.jpdc.2012.01.011.
URL https://doi.org/10.1016/j.jpdc.2012.01.011

[32] C. Kessler, S. Litzinger, J. Keller, Static scheduling of moldable streaming tasks with task fusion for parallel systems with dvfs, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39 (11) (2020) 4166–4178. doi:10.1109/TCAD.2020.3013054.

[33] H. Nishikawa, K. Shimada, I. Taniguchi, H. Tomiyama, Scheduling of moldable fork-join tasks with inter- and intra-task communications, in: Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems, SCOPES '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 7–12. doi:10.1145/3378678.3391875.
URL https://doi.org/10.1145/3378678.3391875

[34] M. Amaris, G. Lucarelli, C. Mommessin, D. Trystram, Generic algorithms for scheduling applications on hybrid multi-core machines, in: 23rd International European Conference on Parallel and Distributed Computing (EuroPar), Santiago de Compostela, Spain, 2017.
URL https://hal.inria.fr/hal-01420798

[35] J. Meng, X. Wu, V. Morozov, V. Vishwanath, K. Kumaran, V. Taylor, Skope: A framework for modeling and exploring workload behavior, in: 11th ACM Conference on Computing Frontiers, CF '14, New York, NY, USA, 2014, pp. 6:1–6:10. doi:10.1145/2597917.2597928.
URL http://doi.acm.org/10.1145/2597917.2597928

[36] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: A unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput. : Pract. Exper. 23 (2) (2011) 187–198. doi:10.1002/cpe.1631.
URL http://dx.doi.org/10.1002/cpe.1631

[37] 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, D. S. Katz, Pegasus: A framework for mapping complex scientific workflows onto distributed systems, Sci. Program. 13 (3) (2005) 219–237.

doi:10.1155/2005/128026.
URL http://dx.doi.org/10.1155/2005/128026

[38] R. Piscitelli, A. D. Pimentel, Design space pruning through hybrid analysis in system-level design space exploration., in: DATE, IEEE, 2012, pp. 781–786.
URL http://dblp.uni-trier.de/db/conf/date/date2012.html

[39] M. Chéramy, P.-E. Hladik, A.-M. Déplanche, SimSo: A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms, in: 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), Madrid, Spain, 2014, p. 6 p.
URL https://hal.archives-ouvertes.fr/hal-01052651

[40] A. D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comput. 55 (2) (2006) 99–112. doi:10.1109/TC.2006.16.
URL http://dx.doi.org/10.1109/TC.2006.16

[41] H. Nikolov, T. Stefanov, E. Deprettere, Systematic and automated multiprocessor system design, programming, and implementation, Trans. Comp.-Aided Des. Integ. Cir. Sys. 27 (3) (2008) 542–555. doi:10.1109/TCAD.2007.911337.
URL http://dx.doi.org/10.1109/TCAD.2007.911337

[42] V. Kelefouras, G. Keramidas, N. Voros, Combining software cache partitioning and loop tiling for effective shared cache management, ACM Trans. Embed. Comput. Syst. 17 (3).

[43] V. Kelefouras, G. Keramidas, N. Voros, Cache partitioning + loop tiling: A methodology for effective shared cache management, in: 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2017, pp. 477–482.

[44] H. Khaleghzadeh, R. R. Manumachu, A. Lastovetsky, A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous hpc platforms, IEEE Transactions on Parallel and Distributed Systems 29 (10) (2018) 2176–2190. doi:10.1109/TPDS.2018.2827055.

[45] H. Zhao, R. Sakellariou, An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm, in: Parallel Processing: 9th International Euro-Par Conference, Springer Berlin Heidelberg, 2003, pp. 189–194.

[46] S. Baskiyar, P. SaiRanga, Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length, in: Proc. Int'l Conf. Parallel Processing Workshops, Vol. 2003, IEEE, 2003, pp. 97– 103.

[47] C. Hui, A high efficient task scheduling algorithm based on heterogeneous multi-core processor, in: 2nd International Workshop on Database Technology and Applications (DBTA), IEEE, 2010. doi:10.1109/DBTA.2010.5659041.

[48] F. Suter, Daggen: A synthethic task graph generator, https://github.com/frs69wq/daggen.

[49] G. Mehta, G. Juve, Workflow generator, https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator.

[50] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Gener. Comput. Syst. 29 (3) (2013) 682–692. doi:10.1016/j.future.2012.08.015.
URL http://dx.doi.org/10.1016/j.future.2012.08.015