Memory-aware list scheduling for hybrid platforms

Julien Herrmann[1], Loris Marchal[1] and Yves Robert[1,2]
1. Ecole Normale Supérieure de Lyon, CNRS & INRIA, France
2. University of Tennessee Knoxville, USA
{julien.herrmann|loris.marchal|yves.robert}@ens-lyon.fr

***Abstract*—This paper provides memory-aware heuristics to schedule tasks graphs onto heterogeneous resources, such as a dual-memory cluster equipped with multicores and a dedicated accelerator (FPGA or GPU). Each task has a different processing time for either resource. The optimization objective is to schedule the graph so as to minimize execution time, given the available memory for each resource type. In addition to ordering the tasks, we must also decide on which resource to execute them, given their computation requirement and the memory currently available on each resource. The major contributions of this paper are twofold: (i) the derivation of an intricate integer linear program formulation for this scheduling problem; and (ii) the design of memory-aware heuristics, which outperform the reference heuristics HEFT and MINMIN on a wide variety of problem instances. The absolute performance of these heuristics is assessed for small-size graphs, with up to 30 tasks, thanks to the linear program.**

I. Introduction

Modern computing platforms are heterogeneous: a typical node is composed of a multi-core processor equipped with a dedicated accelerator, such as an FPGA or a GPU. These two computational units (cores and accelerator) are strongly heterogeneous. To complicates matters, each unit comes with its dedicated memory. Altogether, such an architecture with two computational resources and two memory types, which we call a *dual memory system* hereafter, leads to new challenges when scheduling scientific workflows on such platforms.

In recent work [1], we have introduced a simplified model to assess the complexity of scheduling for dual-memory systems. We have studied various traversals of tree-shaped task graphs, where each task was pre-assigned to one resource type, and where the optimization goal was to minimize the amount of memory of both types. In real-life, there are several complications: (i) tasks are not pre-assigned but can be dynamically assigned to either resource; (ii) task graphs are general DAGs rather then trees; and (iii) one aims at optimizing total execution time (or *makespan*) while minimizing memory usage. However, the simplified model was useful to assert the intrinsic difficulty of the problem: it is NP-complete to decide whether there exists a tree traversal that satisfies bounds on each memory usage: worse, it is impossible to approximate within a constant factor pair both absolute minimum memory amounts. Here the absolute minimum memory of a given type is computed when assuming an infinite amount of memory of the other type. All theses results, although negative, have laid the foundations of scheduling for dual-memory systems.

In this paper, we adopt a pragmatic approach and address the general problem, that of scheduling general tasks graphs on dual-memory systems. The objective is makespan minimization, while enforcing that memory capacities of each type are not exceeded. Given the negative results listed above, there is little hope to derive approximation algorithms. We lower our ambition and aim at designing efficient heuristics for this problem, which we validate through an extensive set of simulations for a variety of scientific benchmarks. However, one major theory-oriented contribution of this paper is the derivation of an Integer Linear Program (ILP) formulation for the general problem. This linear program turns out very intricate, due to expressing all constraints related to memory usage, and it has a large number of variables and constraints. Still, it enables us to determine the optimal solution for small-size problems, up to 30 tasks, and thereby to assess the optimal performance of our heuristics for small instances.

HEFT [2] is widely used for scheduling scientific workflows on heterogeneous resources. It is an extension of critical-path list-scheduling that schedules the current ready task on the resource that will complete its execution as soon as possible (given already taken scheduling decisions). By considering task completion instead of task initiation, HEFT is able to take CPU speed heterogeneity into account. However, HEFT has no provision to optimize memory usage, even for a single-memory system, and a fortiori for a dual-memory one. Another main contribution of this paper is to introduce a memory-aware variant of HEFT for dual-memory systems. Similarly, we design a memory-aware variant of MINMIN [3], another reference heuristic for DAGs where the next task to be executed is selected dynamically (rather than according to some static criteria as in HEFT): MINMIN picks the ready task which has the smallest completion time and executes it on the best available processor.

The rest of the paper is organized as follows. We start with a brief overview of related work in Section II. Then we detail the model and framework in Section III. Section IV is devoted to expressing an optimal schedule in terms of the solution of a complex ILP. We introduce the new heuristics in Section V, and assess their performance through an extensive set of simulations in Section VI. Finally, we provide concluding remarks in Section VII.

II. Related work

A. Task graph scheduling

Computations with dependencies are naturally modeled through task graphs, where nodes represent computational tasks and edges represent dependencies. Task graph scheduling has been the subject of a wide literature, ranging from theoretical studies to practical ones. On the theoretical side, the most used techniques are list scheduling [4], clustering [5], and task duplication [6]. On the practical side, task graphs have been widely used to model complex workflows in grid computing [7]. Scheduling task graphs on grids is the subject of a wide literature, and many tools exist to manage and schedule such workflows, such as MOTEUR [8]. These tools usually include scheduling heuristics to map workflow tasks onto available resources. These heuristics were often inherited from the task graph scheduling literature, and were more or less adapted to cope with the intrinsic heterogeneity of grid environments. The most famous task graph scheduling algorithm for grids and heterogeneous platforms is HEFT [2], which we use and adapt to our dual-memory context.

B. Scheduling with memory constraints

The problem of scheduling a task graph under memory constraints appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics or geophysical simulations. The problem of task graphs handling large data has been identified in [9] which proposes some simple heuristic solutions. Most existing theoretical studies are restricted to tree-shaped task graphs, that arise in some application domains such as the factorization of sparse matrices using the multifrontal method [10], [11]. We refer the interested reader to our recent paper [1] for an extended bibliography on adding memory constraints to the problem of scheduling tree-shaped task graphs.

C. Hybrid computing

Hybrid computing consists in the simultaneous use of CPUs and GPUs to optimize performance for high performance computing. Since CPUs and GPUs are powerful for specific and different tasks, its is natural to schedule tasks on their "favorite" resource, that is, the resource where their execution time is minimal. This has successfully been achieved to increase performance in linear algebra libraries [12], [13]. There also exist software tools that schedule an application composed of tasks with both CPU and GPU implementations on hybrid platforms: for instance, StarPU [14] optimizes the execution time of an application by scheduling its tasks on multiple kinds of resources, based on predictions of execution and data transfer times.

III. Model and framework

As stated above, we deal with general task graph traversals on a dual-memory system, where each task can be executed on either of the two processing units, that is, with its associated data in one of either memory. Dependencies are in the form of input and output files: each task accepts a set of files as input from each of its parent nodes in the DAG, and produces a set of files to be consumed by each child node. We start this section by formally writing all the constraints that need to be satisfied during a traversal. Finally, we state the target optimization problem in Section III-C.

A. Flow and resources constraints

We consider, in this paper, a dual-memory heterogeneous platform with P_1 identical processors which share the first memory and with P_2 identical processors which share the second memory. For clarity, in the rest of the paper, the first memory will be referred to as the *blue* memory and the P_1 processors sharing it will be called the *blue* processors. Similarly, the second memory and its processors will be associated to the color *red* as depicted in Figure 1.

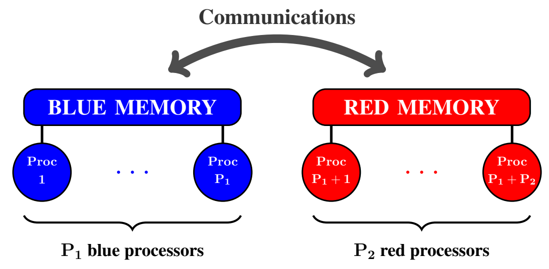

Figure 1: Description of the dual-memory platform.

The application is described by a Directed Acyclic Graph $\mathcal{D} = (V, E)$ composed of $|V| = n$ nodes, or tasks, numbered from 1 to n. We let *Children*$(i) = \{j \in V \text{ s.t. } (i,j) \in E\}$ denote the set of the children of i and *Parents*$(i) = \{j \in V \text{ s.t. } (j,i) \in E\}$ denotes the set of the parents of i. Dependencies imply a topological order, where a parent node has to be processed before its children. Here are some definitions:

- Each task i in the DAG requires a processing time of $W_i^{(1)}$ on one of the *blue* processors and a processing time of $W_i^{(2)}$ on one of the *red* processors.
- Each communication $(i,j) \in E$ is instantaneous if nodes i and j are executed on processors that belong to the same memory. Otherwise, the file produced by node i and needed as input by node j has to be sent from one memory to the other. This transfer takes $C_{i,j}$ time units.

For example, consider the toy example DAG $\mathcal{D}_{ex}$ depicted in Figure 2. Task T_1 can be processed in $W_1^{(1)} = 3$ time units on a *blue* processor and in $W_1^{(2)} = 1$ time unit on a *red* processor. If tasks T_1 and T_2 are not executed on the same memory, the communication (T_1, T_2) will take $C_{1,2} = 1$ time unit to be processed. We point out that all communication times are set arbitrarily to 1 in this example (e.g., to account for a high start-up cost). Of course, an affine formula (such as $C_{i,j} = \alpha + \beta F_{i,j}$), or even arbitrary values, can be used in the model.

Given an application DAG, our goal is to determine where each task should be executed (the allocation) and at what time each task and communication may be started (the starting times). The allocation is described by function $proc : V \to [\![1, P_1 + P_2]\!]$ where $\forall i \in V$, $proc(i)$ represents the index of the processor that processes task i. $proc(i) \leq P_1$ represents a *blue* processor while $proc(i) > P_1$ represents a *red* processor. The starting times are expressed as two functions $\sigma : V \to \mathbb{R}^+$ and $\tau : E \to \mathbb{R}^+$ where $\forall i \in V$, $\sigma(i)$ represents the starting time of task i and $\forall (i,j) \in E$, $\tau(i)$ represents the starting time of communication (i,j).

Let W_i be the actual processing time of task i in the schedule $s = (\sigma, \tau, proc)$:

$$W_i = \begin{cases} W_i^{(1)} & \text{if } proc(i) \leq P_1 \\ W_i^{(2)} & \text{otherwise} \end{cases}$$

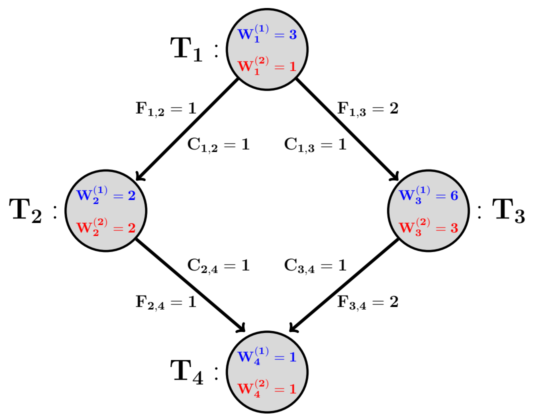
T_1 : $W_1^{(1)}=3$ $W_1^{(2)}=1$
T_2 : $W_2^{(1)}=2$ $W_2^{(2)}=2$
$W_3^{(1)}=6$ $W_3^{(2)}=3$: T_3
T_4 : $W_4^{(1)}=1$ $W_4^{(2)}=1$
$F_{1,2}=1$
$F_{1,3}=2$
$C_{1,2}=1$
$C_{1,3}=1$
$C_{2,4}=1$
$C_{3,4}=1$
$F_{2,4}=1$
$F_{3,4}=2$

Figure 2: Description of $\mathcal{D}_{ex}$.

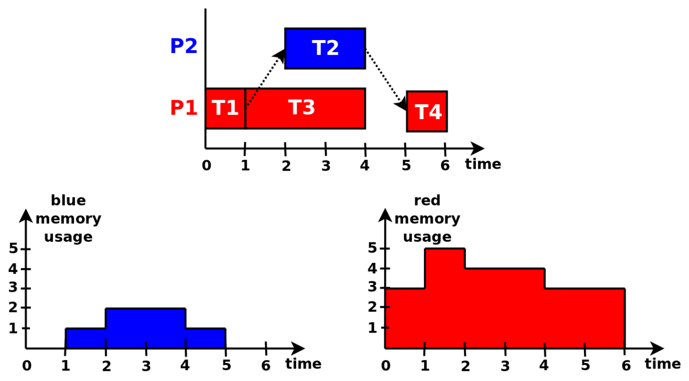

Figure 3: Representation of schedule s_1 for $\mathcal{D}_{ex}$.

We note $COMM_{i,j}$ the actual time taken by communication (i,j) in the schedule $s=(\sigma,\tau,proc)$:

$$COMM_{i,j} = \begin{cases} 0 & \text{if } proc(i) \leq P_1 \text{ and } proc(j) \leq P_1 \\ 0 & \text{if } proc(i) > P_1 \text{ and } proc(j) > P_1 \\ C_{i,j} & \text{otherwise} \end{cases}$$

A schedule $s=(\sigma,\tau,proc)$ of $\mathcal{D}$ is a *valid schedule* if it respects:

- flow dependencies, $\forall (i,j) \in E$:

$$\begin{cases} \sigma(i) + W_i \leq \tau(i,j) \\ \tau(i,j) + COMM_{i,j} \leq \sigma(j) \end{cases}$$

- resource constraints, $\forall (i,j) \in V^2$:

$$proc(i) = proc(j) \implies \begin{cases} \sigma(i) \leq \sigma(j) + W_j \\ \text{or} \\ \sigma(j) \leq \sigma(i) + W_i \end{cases}$$

The makespan of the schedule is the finish time of the last task:

$$Makespan = \max_{i \in V} (\sigma(i) + W_i)$$

Back to the example $\mathcal{D}_{ex}$, on a dual-memory platform with one *blue* processor and one *red* processor ($P_1 = P_2 = 1$), consider the following schedule s_1 depicted in Figure 3 for :

$$\begin{cases} \sigma_1(T_1) = 0, \quad \sigma_1(T_2) = 2, \quad \sigma_1(T_3) = 1, \quad \sigma_1(T_4) = 5 \\ \tau_1(T_1, T_2) = 1, \quad \tau_1(T_2, T_4) = 4 \\ proc_1(T_1) = 2, \quad proc_1(T_2) = 2, \quad proc_1(T_3) = 1 \\ proc_1(T_4) = 2 \end{cases}$$

Schedule $s_1 = (\sigma_1, \tau_1, proc_1)$ is a valid schedule for $\mathcal{D}_{ex}$, with $Makespan = 6$.

B. Memory constraints

As stated above, in our model, the dependencies are in the form of input and output files. Each node i in the DAG has an input file of size $F_{j,i}$ for each $j \in Parents(i)$. If i is not the root, its input file is produced by its parents; if i is the root, then $Parents(i) = \emptyset$ and its input files may be of null size, or it may receive input from the outside world. Each non-terminal node i in the DAG, when executed, produces a file of size $F_{i,j}$ for each $j \in Children(i)$. If i is a terminal node, then $Children(i) = \emptyset$ and i produces a file of null size (we consider that terminal data produced by terminal nodes are directly sent to the outside world).

During the processing of a task i on one of the processors, the memory on which this processor operates must contain all the input and output files. The amount of memory $MemReq(i)$ that is needed for this processing is thus:

$$MemReq(i) = \left(\sum_{j \in Parents(i)} F_{j,i} \right) + \left(\sum_{j \in Children(i)} F_{i,j} \right)$$

For instance, in $\mathcal{D}_{ex}$, $MemReq(T_3) = F_{1,3} + F_{3,4} = 4$. Note that the memory needed for the execution of the task itself can easily be accounted for, by adding a fictitious parent task. After task i has been processed, its input files are discarded, while its output files are kept in memory until the processing of its children. Thus, for a schedule $s = (\sigma, proc)$ of $\mathcal{D}$, if a node i is processed by a *blue* processor, the actual amount of *blue* memory used to process the node i is:

$$BlueMemUsed(s,i) = \left(\sum_{j \in Children(i)} F_{i,j} \right) + \sum_{e \in S_{blue}} F_e$$

where S_{blue} denotes the set of files (represented by the edges of $\mathcal{D}$) stored in the *blue* memory, when the scheduler decides to execute task i. Note that S_{blue} must contain the input files of task i. After the processing of node i, we have:

$$S_{blue} \leftarrow (S_{blue} \backslash \{(j,i), j \in Parents(i)\}) \\ \cup \{(i,j), j \in Children(i)\}$$

Of course, the same holds for $RedMemUsed$ and S_{red} if i happens to be processed by a *red* processor. Initially, the input file of the root is arbitrarily located in S_{blue}.

Consider the schedule s_1 depicted in Figure 3. The execution of task T_1 uses $RedMemUsed(T_1) = F_{1,2} + F_{1,3} = 3$ units of *red* memory. The execution of task T_2 uses $BlueMemUsed(T_2) = F_{1,2} + F_{2,4} = 2$ units of *blue* memory. The execution of task T_3 uses $RedMemUsed(T_3) = F_{1,2} + F_{1,3} + F_{3,4} = 5$ units of *red* memory. And the execution of task T_4 uses $RedMemUsed(T_4) = F_{2,4} + F_{3,4} = 3$ units of *red* memory.

Each time there is a data dependence between two tasks assigned to different memories, the output file of the source task has to be loaded from one memory into the other. During the processing of the communication (i, j), both memories contain the file of size $F_{i,j}$ being copied. Thus, for instance, if i has been assigned on a *blue* processor and j has been assigned on a *red* processor, the amount of *blue* and *red* memory needed for this processing is $F_{i,j}$:

$$\mathit{BlueMemReq}(i, j) = F_{i,j}, \quad \mathit{RedMemReq}(i, j) = F_{i,j}$$

After the communication has been processed, the input file from the *blue* memory is discarded, while the output file is kept in the *red* memory until the processing of j. Thus, for a schedule $s = (\sigma, \mathit{proc})$ of $\mathcal{T}$, the actual amounts of memory used to process the communication (i, j) are:

$$\mathit{BlueMemUsed}(s, (i, j)) = F_{i,j} + \sum_{e \in S_{blue} \setminus \{(i,j)\}} F_e$$

$$\mathit{RedMemUsed}(s, (i, j)) = F_{i,j} + \sum_{e \in S_{red}} F_e$$

Note that S_{blue} must contain the input file of task i. After the processing of the communication (i, j) we have:

$$S_{blue} \leftarrow S_{blue} \setminus \{(i, j)\}, \qquad S_{red} \leftarrow S_{red} \cup \{(i, j)\}$$

It is important to state that communication (i, j) does not need to be fired right after the execution of task i. The only constraint is that the processing of communication (i, j) must follow the execution of i and precede the execution of j. This flexibility in the schedule severely complicates the search for efficient traversals.

C. Optimization problem

As stated above, we face an optimization problem under memory constraints. The *memory peak* is the maximum usage of each memory over the whole schedule $s = (\sigma, \mathit{proc})$ of the DAG $\mathcal{D}$, and is defined for the *blue* and the *red* memory by:

$$M^s_{\text{blue}}(\mathcal{D}) = \max_i \ \mathit{BlueMemUsed}(s, i)$$
$$M^s_{\text{red}}(\mathcal{D}) = \max_i \ \mathit{RedMemUsed}(s, i)$$

In practical settings, the amount of memory at disposal is limited. Let note $M^{(blue)}$ and $M^{(red)}$ the bounds on the *blue* and the *red* memories. We aim at finding the optimal schedule $s_{opt}(M^{(blue)}, M^{(red)})$ of the DAG $\mathcal{D}$, defined as the schedule with minimal makespan among all schedules s that does not require more memory than available, i.e., that enforce the bounds on memory peaks: $M^s_{\text{blue}}(\mathcal{T}) \le M^{(blue)}$ and $M^s_{\text{red}}(\mathcal{T}) \le M^{(red)}$. From [1], we know that this problem is NP-complete, even without any makespan constraint.

Back to the schedule s_1 described in Figure 3, assume a dual-memory platform with one *blue* processor and one *red* processor. We compute that s_1 uses $M^{s_1}_{\text{blue}}(\mathcal{D}_{ex}) = 2$ units of *blue* memory and $M^{s_1}_{\text{red}}(\mathcal{D}_{ex}) = 5$ units of *red* memory. If we set the memory bounds $M^{(blue)} = M^{(red)} = 5$, it is clear that s_1 is the optimal schedule. But if we set $M^{(blue)} = M^{(red)} = 4$, s_1 is no longer an acceptable schedule. In this case, the optimal schedule for $\mathcal{D}_{ex}$ will be s_2, the schedule depicted in Figure 4.

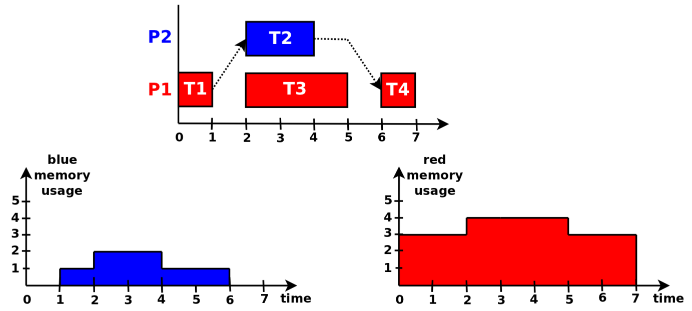

Figure 4: Representation of schedule s_2

Schedule s_2 has a smaller memory peak than s_1 but has a larger $\mathit{Makespan} = 7$. This small example illustrates the necessary tradeoff between memory and makespan.

IV. ILP FORMULATION

In this section, we describe how to compute an optimal schedule σ_{opt} through a computationally expensive ILP (Integer Linear Program). The objective is twofold: (i) to provide an optimal solution for small instances and (ii) to compare the heuristics presented in the following section with the optimal schedule, to evaluate their absolute quality.

Our approach is motivated by the successful attempt to derive such an ILP formulation for several variants of the DAG scheduling problems, such as [15], [16]. However, to the best of our knowledge, none of the existing ILP handles the memory usage of the schedule. A major contribution of this paper is the introduction of additional constraints that enforce memory constraints, as those described in Section III-B.

The variables used by our linear program are listed in Figure 5. The t_i's and τ_{ij}'s variables represent the starting time of the tasks and of the communications. M is the makespan value to minimize. The p_i's and b_i's variables describe the allocation of task i on the resources and are used to compute the value of the w_i's variables, which represent the actual computing time of task i. The ϵ_{ij}'s and δ_{ij}'s variables are used to enforce resources constraints. Finally, to compute the amount of memory used by the schedule at any time, we need to know the order in which all tasks and communications are processed. This is achieved through variables σ_{ij}, σ'_{kij}, m_{ij}, m'_{kij}, c_{ijk}, c'_{ijkp}, d_{ijk} and d'_{ijkp}. These numerous variables are needed to ensure that the schedule is properly defined, and that we precisely know which tasks are processed and which data are present in a given memory at any time, to ensure that the memory usage is kept below the prescribed bound.

Due to the numerous variables that describe a schedule, the linear program counts a large number of constraints to ensure that these variables correspond to their definition given in Figure 5. For the sake of completeness, we give the whole linear program in Figure 6, and we detail the most significant constraints below.

Constraints (1) to (25) describes a schedule of the DAG onto the heterogeneous platform, and have nothing to do with memory constraints. They also ensure that communication times are respected when a data needs to be moved from

M	makespan of the corresponding schedule
t_i	starting time of task i
τ_{ij}	starting time of communication (i,j)
p_i	index of the processor where the task i is to be executed
b_i	equal to 0 if task i is executed on the *red* memory and 1 if it is executed on the *blue* memory
w_i	actual computing time of task i in the corresponding schedule
ϵ_{ij}	equal to 1 if the processor index of task i is strictly less than that of task j and 0 otherwise
δ_{ij}	equal to 1 if task i and task j are executed on the same memory and 0 otherwise
σ_{ij}	1 if task i finishes before task j starts and 0 otherwise
σ'_{kij}	equal to 1 if task k finishes before communication (i,j) starts and 0 otherwise
m_{ij}	equal to 1 if task i starts before task j starts and 0 otherwise
m'_{kij}	equal to 1 if task k starts before communication (i,j) starts and 0 otherwise
c_{ijk}	equal to 1 if communication (i,j) starts before task k starts and 0 otherwise
c'_{ijkp}	equal to 1 if communication (i,j) starts before communication (k,p) starts and 0 otherwise
d_{ijk}	equal to 1 if communication (i,j) finishes before task k starts and 0 otherwise
d'_{ijkp}	equal to 1 if communication (i,j) finishes before communication (k,p) starts and 0 otherwise

Figure 5: Variables of the linear program

$$
\begin{array}{rll}
\min_{t,p,\sigma,\epsilon} & M & \\
\forall i \in V, & t_i + w_i \le M & (1) \\
\forall (i,j) \in E, & t_i + w_i \le \tau_{ij} & (2) \\
\forall (i,j) \in E, & \tau_{ij} + (1-\delta_{ij})C_{ij} \le t_j & (3) \\
\forall i \neq j \in V, & t_j - t_i - m_{ij}M_{max} \le 0 & (4a) \\
\forall i \neq j \in V, & t_j - t_i + (1-m_{ij})M_{max} \ge 0 & (4b) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k - m'_{kij}M_{max} \le 0 & (5a) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k + (1-m'_{kij})M_{max} \ge 0 & (5b) \\
\forall i \neq j \in V, & t_j - t_i - w_i - \sigma_{ij}M_{max} \le 0 & (6a) \\
\forall i \neq j \in V, & t_j - t_i - w_i + (1-\sigma_{ij})M_{max} \ge 0 & (6b) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k - w_k - \sigma'_{kij}M_{max} \le 0 & (7a) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k - w_k + (1-\sigma'_{kij})M_{max} \ge 0 & (7b) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} - c_{ijk}M_{max} \le 0 & (8a) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} + (1-c_{ijk})M_{max} \ge 0 & (8b) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} - c'_{ijkp}M_{max} \le 0 & (9a) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} + (1-c'_{ijkp})M_{max} \ge 0 & (9b) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} - (1-\delta_{ij})C_{ij} - d_{ijk}M_{max} \le 0 & (10a) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} - (1-\delta_{ij})C_{ij} + (1-d_{ijk})M_{max} \ge 0 & (10b) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} - (1-\delta_{ij})C_{ij} - d'_{ijkp}M_{max} \le 0 & (11a) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} - (1-\delta_{ij})C_{ij} + (1-d'_{ijkp})M_{max} \ge 0 & (11b) \\
\forall i,j \in V, & p_j - p_i - \epsilon_{ij}|P| \le 0 & (12a) \\
\forall i \neq j \in V, & p_j - p_i - 1 + (1-\epsilon_{ij})|P| \ge 0 & (12b) \\
\forall i \in V, & p_i - |P_0| - |P|b_i \le 0 & (13a) \\
\forall i \in V, & p_i - |P_0| - 1 + (1-b_i)(|P|+1) \ge 0 & (13b) \\
\forall i,j \in V, & m_{ij} + m_{ji} \ge 1 & (14) \\
\forall i,j \in V, & \sigma_{ij} + \sigma_{ji} \le 1 & (15) \\
\forall (i,j) \in E, \forall k \in V, & m'_{kij} + c_{ijk} \ge 1 & (16) \\
\forall (i,j),(k,p) \in E, & c'_{ijkp} + c'_{kpij} \ge 1 & (17) \\
\forall (i,j),(k,p) \in E, & d'_{ijkp} + d'_{kpij} \le 1 & (18) \\
\forall i \in V, \forall k \in V, & m_{ik} \ge \sigma_{ik} & (19) \\
\forall (i,j) \in E, \forall k \in V, & \sigma_{ik} \ge c_{ijk} & (20) \\
\forall (i,j) \in E, \forall k \in V, & c_{ijk} \ge d_{ijk} & (21) \\
\forall (i,j) \in E, \forall k \in V, & d_{ijk} \ge m_{jk} & (22) \\
\forall i,j \in V, & \delta_{ij} \le 1 + b_i - b_j, \delta_{ij} \le 1 + b_j - b_i, & \\
 & \delta_{ij} \ge b_i + b_j - 1 \text{ and } \delta_{ij} \ge 1 - b_i - b_j & (23) \\
\forall i \in V, & w_i \ge b_i W_i^{(2)} + (1-b_i)W_i^{(1)} & (24a) \\
\forall i \in V, & w_i \le b_i W_i^{(2)} + (1-b_i)W_i^{(1)} & (24b) \\
\forall i \neq j \in V, & \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \ge 1 & (25) \\
\forall i \in V, & \sum_{(k,p)\in E}(\delta_{ik}(m_{ki} - d_{kpi}) + \delta_{ip}(c_{kpi} - \sigma_{pi}))F_{kp} & \\
 & \le b_i M_{blue} + (1-b_i)M_{red} & (26) \\
\forall (i,j) \in E, & \sum_{(k,p)\in E}(\delta_{kj}(m'_{kij} - d'_{kpij}) + \delta_{pj}(c'_{kpij} - \sigma'_{pij}))F_{kp} & \\
 & \le b_j M^{(blue)} + (1-b_j)M^{(red)} + \delta_{ij}M_{max} & (27)
\end{array}
$$

Figure 6: Constraints of the ILP.

one memory to another. Here is a short description of these constraints:

- Constraint (1) ensures that variable M representing the makespan will be larger than or equal to the completion time of the last task.
- Constraint (2) ensures that communication (i,j) starts after the completion of task i.
- Constraints (3) ensures that task j starts after the completion of every possible communication (i,j). We can note that, since $\delta_{ij} = 1$ if and only if task i and j are executed on the same memory, $(1-\delta_{ij})C_{ij}$ is the actual processing time of communication (i,j).
- In Constraints (4a) and some of the following ones, we need an upper bound M_{max} on the possible value of M. This bound is set arbitrarily to $M_{max} = \sum_{i\in V} W_i^{(1)} + \sum_{i\in V} W_i^{(2)} + \sum_{(i,j)\in E} C_{i,j}$. Constraints (4a), (4b) and (14) ensure that $m_{i,j}$ and $m_{j,i}$ are correctly defined: $m_{i,j} = 1$ if $t_j > t_i$, $m_{i,j} = 0$ if $t_j > t_i$ and if $t_j = t_i$, at least one between m_{ij} and m_{ji} is equal to 1. This is important when computing the amount of memory in Constraint (26).
- Similarly Constraints (5a) to (18) ensure that m'_{kij}'s, σ_{ij}'s, σ'_{kij}'s, c_{ijk}'s, c'_{ijkp}'s, d_{ijk}'s, d'_{ijkp}'s, ϵ'_{ij}'s and b_i's variables are well defined.
- Constraint(19) ensures that task ordering is defined consistently, even for tasks with zero processing time (such tasks will appear when pipelining communications in Section VI).
- Constraint (20) ensures that if communication (i,j) starts before task k starts, task i must finish before task k starts. Constraints (21) and (22) ensure that the linear program defines a valid schedule for communications and tasks.
- Constraints (23) ensures that δ_{ij}'s variables are well defined, i.e., $\delta_{ij} = 1$ if and only if $b_i = b_j$.
- Constraints (24a) and (24b) ensure that w_i's variables are well defined, i.e., $w_i = W_i^{(1)}$ if and only if $b_i = 0$ and $w_i = W_i^{(2)}$ if and only if $b_i = 1$.
- Constraint (25) represents resource constraints as seen in Section III-A: if two tasks are running at the same time, they are not on the same processor.

Finally, Constraint (26) deals with memory constraints, and ensures that the model defined in Section III-B is observed at the beginning of each task i. Specifically, $b_i M_{blue} + (1-b_i)M_{red}$ is the memory bound on the memory on which task i is executed. When i is started, we ensure that the sum of the files stored in the corresponding memory when we start task i is smaller than this bound. We claim that $\forall (k,p) \in E$, the file of size F_{kp} will be in the corresponding memory when task i starts if and only if either "task i and task k are in the same memory and we started task k but communication (k,p) is not finished yet" or "task i and task p are in the same memory and we started communication (k,p) but task p is not finished yet". This explains Constraint (26). Similarly Constraint (27) ensures that the memory constraint is respected at the beginning of every communication (i,j).

Constraints (26) and (27) are not linear. However, they can be linearized using the technique presented in [15], [16]. To do so, we introduce the variables $\alpha_{kpi} = \delta_{ik}(m_{ki} - d_{kpi})$, $\beta_{kpi} = \delta_{ip}(c_{kpi} - \sigma_{pi})$, $\alpha'_{kpi} = \delta_{kj}(m'_{kij} - d'_{kpij})$ and $\beta'_{kpij} =$

$\delta_{pj}(c'_{kpij} - \sigma'_{pij})$. Constraints (26) and (27) are then replaced by the constraints in Figure 7.

$$\begin{array}{rll}
\forall i \in V, & \sum_{(k,p)\in E}(\alpha_{kpi} + \beta_{kpi})F_{kp} & \\
& \le b_i M^{(red)} + (1-b_i)M^{(blue)} & (26) \\
\forall i \in V, \forall (k,p) \in E, & \alpha_{kpi} \ge \delta_{ik} + m_{ki} - d_{kpi} - 1 & (26a) \\
\forall i \in V, \forall (k,p) \in E, & 2\alpha_{kpi} \le \delta_{ik} + m_{ki} - d_{kpi} & (26b) \\
\forall i \in V, \forall (k,p) \in E, & \beta_{kpi} \ge \delta_{ip} + c_{kpi} - \sigma_{pi} - 1 & (26c) \\
\forall i \in V, \forall (k,p) \in E, & 2\beta_{kpi} \le \delta_{ip} + c_{kpi} - \sigma_{pi} & (26d) \\
\forall (i,j) \in E, & \sum_{(k,p)\in E}(\alpha'_{kpij} + \beta'_{kpij})F_{kp} & \\
& \le b_i M^{(red)} + (1-b_i)M^{(blue)} + \delta_{ij} M_{max} & (27) \\
\forall (i,j) \in E, \forall (k,p) \in E, & \alpha'_{kpij} \ge \delta_{kj} + m'_{kij} - d'_{kpij} - 1 & (27a) \\
\forall (i,j) \in E, \forall (k,p) \in E, & 2\alpha'_{kpij} \le \delta_{kj} + m'_{kij} - d'_{kpij} & (27b) \\
\forall (i,j) \in E, \forall (k,p) \in E, & \beta'_{kpij} \ge \delta_{pj} + c'_{kpij} - \sigma'_{pij} - 1 & (27c) \\
\forall (i,j) \in E, \forall (k,p) \in E, & 2\beta'_{kpij} \le \delta_{pj} + c'_{kpij} - \sigma'_{pij} & (27d)
\end{array}$$

Figure 7: Linearization of the last two constraints of the ILP.

For an arbitrary DAG $\mathcal{D} = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, the ILP has $O(m^2 + mn)$ variables and $O(m^2 + mn)$ constraints.

V. Heuristics

Given the complexity of optimizing the makespan under memory constraints, we propose two heuristics in this section, MemHEFT and MemMinMin. The key idea is to add memory awareness to the design of traditional scheduling heuristics.

A. *The* MemHEFT *algorithm*

MemHEFT is based on HEFT (Heterogeneous Earliest Finish Time) [2]. The HEFT algorithm is highly competitive and widely used to schedule static DAGs on heterogeneous platforms with a low time complexity. HEFT has two major phases: a *task prioritizing phase* for computing the priorities of all tasks , and a *processor selection phase* for allocating each task (in the order of their priorities) to their best processor, defined as the one which minimizes the task finish time.

The MemHEFT algorithm follows the same pattern as HEFT. In our model, there are only two processor types, hence each selected task will be mapped on one of two candidates, namely the processors with earliest available time in each type. In other words, the *processor selection phase* can be renamed as the *memory selection phase*. In addition, MemHEFT checks memory usage, as explained below.

Task prioritizing phase. This phase is the same as in HEFT and requires the priority of each task to be set with the upward rank value, $rank(i)$, which is based on mean computation and mean communication costs:

$$\forall i \in V,\ rank(i) = \frac{W_i^{(red)} + W_i^{(blue)}}{2} + \max_{j \in Children(i)}\{rank(j) + \frac{C_{i,j}}{2}\}$$

where $Children(i)$ denotes the immediate successors of task i. The task list is generated by sorting the tasks by non-increasing order of $rank(i)$. Tie-breaking is done randomly.

Memory selection phase. For each selected task i and for each memory $\mu \in \{red, blue\}$, we have to compute $EST^{(\mu)}(i)$ the earliest execution start time of task i on memory μ (derived from a given partial schedule). This earliest execution start time has to take into account (i) resource, (ii) precedence, and (iii) memory constraints.

From a resource perspective, task i can not be executed on memory μ before one of the processors operating on memory μ is available. Thus $resource\_EST^{(\mu)}(i)$, the earliest start time of task i on memory μ from a resource point of view, can be expressed as:

$$resource\_EST^{(\mu)}(i) = \min_{proc \text{ in } \mu \text{ mem}}\{avail[proc]\}$$

where $avail[proc]$ is the finish time of the last task assigned to $proc$ in the partial schedule.

From a precedence perspective, all immediate predecessors $j \in Parents(i)$ of task i must have been scheduled. Thus $precedence\_EST^{(\mu)}(i)$, the earliest start time of task i on memory μ from a precedence point of view, is expressed as:

$$precedence\_EST^{(\mu)}(i) = \max_{j \in Parents(i)}\{AFT(j) + \delta_j^{(\mu)} C_{j,i}\}$$

where $\delta_j^{(\mu)} = 0$ is task j is executed on memory μ, 1 otherwise, and $AFT(j)$ is the actual finish time of task j in the partial schedule.

From a memory perspective, we have to keep trace of the memory consumption of our schedule to ensure that it does not violate the memory constraints. Thus, the MemHEFT algorithm maintains for each memory μ the function $free\_mem^{(\mu)}(t)$ that represents the amount of the μ memory available at time t in the partial schedule. Here $free\_mem^{(\mu)}$ is a staircase function (the definition space $\mathbb{R}$ can be partitioned in a finite number of intervals where $free\_mem^{(\mu)}$ is constant) that can be stored as a list of couples $[(x_1, val_1), .., (x_\ell, val_\ell)]$ such that:

$$\forall i \in [1, \ell - 1],\ \forall t \in [x_i, x_{i+1}[,\ free\_mem^{(\mu)}(t) = val_i$$

and $\forall t \ge x_\ell,\ free\_mem^{(\mu)}(t) = val_\ell$. Note that val_ℓ can be non-zero since the partial schedule may keep some files $F_{i,j}$ stored in the memories if task i has been scheduled but task j has not. Thus, to process task i on memory μ at time t without violating memory constraints, there must be enough available memory to store all the input files of task i that were not stored on memory μ yet, and all its output files. Thus, the earliest start time of task i on memory μ from the memory point of view can be expressed as:

$$task\_mem\_EST^{(\mu)}(i) = min\ \{t,\ \text{such that } \forall t' \ge t,$$
$$free\_mem^{(\mu)}(t') \ge \sum_{j \in Parents(i)} (1 - \delta_j^{(\mu)})F_{j,i} + \sum_{j \in Children(i)} F_{i,j}\}$$

If $free\_mem^{(\mu)}$ is stored as a list of size ℓ, $task\_mem\_EST^{(\mu)}(i)$ can be computed in time $O(\ell)$.

The MemHEFT algorithm enforces that when a task i is assigned to the memory μ, every communication $(j, i) \in E$ such that $\delta_j^{(\mu)} = 0$ will start as late as possible, and they will all have a processing time $C_i^{(\mu)} = \max_{(j,i)\in E}\{(1 - \delta_j^{(\mu)})C_{i,j}\}$. Thus, to process task i on memory μ, the earliest start time of every communication $(j, i) \in E$ from the memory point of view can be expressed as:

$$comm\_mem\_EST^{(\mu)}(i) = min\ \{t,\ \text{such that } \forall t' \ge t,$$
$$free\_mem^{(\mu)}(t') \ge \sum_{j \in Parents(i)} (1 - \delta_j^{(\mu)})F_{j,i}\}$$

If $free\_mem^{(\mu)}$ is stored as a list of size ℓ, $comm\_mem\_EST^{(\mu)}(i)$ can be computed in time $O(\ell)$.

Finally, the earliest execution start time of task i on memory μ will be expressed as:

$$\begin{aligned} EST^{(\mu)}(i) = max\ \{ & resource\_EST^{(\mu)}(i), \\ & precedence\_EST^{(\mu)}(i), \\ & task\_mem\_EST^{(\mu)}(i), \\ & comm\_mem\_EST^{(\mu)}(i) + C_i^{(\mu)} \} \end{aligned}$$

The selected task i is assigned to the memory μ_{min} that minimizes its earliest finish time $EFT^{(\mu)}(i) = EST^{(\mu)}(i) + W_i^{(\mu)}$ and then, to the $proc$ that minimizes the idle time $EST(i, \mu_{min}) - avail\_proc(proc)$.

B. *The* MEMMINMIN *algorithm*

The MEMMINMIN algorithm does not include a task prioritizing phase but dynamically decides the order in which tasks are mapped onto resources. It is the memory-aware counterpart of the MINMIN heuristic [3]. Indeed, at each step, MEMMINMIN maintains the set $available\_tasks$ representing the tasks whose predecessors have already been scheduled. Then it selects the task $i_{\min}\ in available\_tasks$ and the memory $\mu_{\min} \in \{red, blue\}$ that minimizes $EFT^{(\mu)}(i)$ as defined in Section V-A (computed from a partial schedule).

For a DAG $\mathcal{D}$ with $|V| = n$ nodes and $|E| = m$ edges, both heuristics have a worst-case complexity of $O(n^2(n+m))$. Their pseudo-code is available in [**?**].

VI. SIMULATION RESULTS

In this section, we conduct several simulations to compare the two heuristics MEMHEFT and MEMMINMIN proposed in Section V, and to assess their absolute performance w.r.t. to the (optimal) ILP solution (Section IV). For each heuristic, we compute its makespan for various amounts of the available *blue* and *red* memories. The heuristics have been implemented in Python 2.7. Source code for all the algorithms, heuristics and simulations is publicly available at http://perso.ens-lyon.fr/julien.herrmann/. The optimal makespan for small graphs has been computed by solving the ILP using the IBM® ILOG® CPLEX® Interactive Optimizer 12.5.0.0.

A. *Experimental setup*

We use four different sets of DAGs: (i) two synthetic sets (randomly generated) of different sizes ,SMALLRANDSET, and LARGERANDSET; and (ii) two applicative sets (from linear algebra benchmarks), LUSET and CHOLESKYSET.

1) Random task graphs: The first and second sets are random DAGs, generated using the Directed Acyclic Graph GENerator (DAGGEN)[1]. DAGGEN uses four popular parameters to define the shape of the DAG: *size*, *width*, *density* and *jumps*.

- The *size* determines the number of node in the DAG. Nodes are organized in levels.

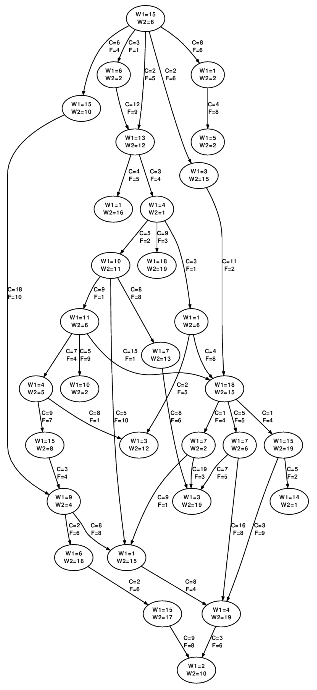

Figure 8: One DAG in SMALLRANDSET.

- The *width* determines the maximum parallelism in the DAG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value to "fork-join" graphs.
- The *density* denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value to many edges.
- Finally random edges are added that go from level l to levels $l+1 \dots l+jumps$.

The first two parameters take values between 0 and 1. This DAG generation procedure is similar to the one used in [17].

SMALLRANDSET is a set of 50 randomly generated DAGs using values $size = 30$, $width = 0.3$, $density = 0.5$ and $jumps = 5$. Then, for each node, the values $W_i^{(1)}$ and $W_i^{(2)}$ are randomly chosen between 1 and 20 and, for each edge, the values $C_{i,j}$ and $F_{i,j}$ are randomly chosen between 1 and 10. One graph of SMALLRANDSET is depicted in Figure 8.

LARGERANDSET is a set of 100 randomly generated DAGs using values $size = 1000$, $width = 0.3$, $density = 0.5$ and $jumps = 5$. Then, for each node and each edge, the values $W_i^{(1)}$, $W_i^{(2)}$, $C_{i,j}$ and $F_{i,j}$ are randomly chosen between 1 and 100. One graph of LARGERANDSET is depicted in Figure 9.

2) Linear algebra task graphs: The third and four sets contain representative DAGs from dense linear algebra kernels.

LUSET contains DAGs representing the task graph of the LU factorization of a tiled square matrix. At each step of this factorization, the diagonal tile is factored with a GETRF kernel, the first row and the first column of tiles are eliminated with a TRSM kernel, and the remaining tiles are updated with a GEMM kernel. Another step of the LU factorization is then applied on the trailing matrix involving a workflow dependencies among the kernels working on the same tiles.

[1]The code for the generator is publicly available at https://github.com/frs69wq/daggen.

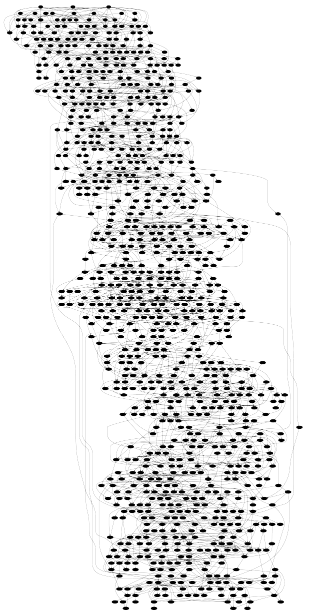

Figure 9: One DAG in LARGERANDSET.

CHOLESKYSET contains DAGs representing the task graph of the Cholesky factorization of tiled symmetrical matrix. At each step of this factorization, the diagonal tile is factored with a POTRF kernel, the first column and the diagonal of tiles are processed with respectively a TRSM and a SYRK kernel, and the remaining tiles are updated with a GEMM kernel. Another step of the Cholesky factorization is then applied on the trailing matrix involving a workflow dependencies among the kernels working on the same tiles.

More details on the tiled LU and Cholesky factorizations can be found in [18]. The classic DAGs of both factorizations do not exactly fit our model, as the output of a node (typically the kernel used for factoring a diagonal tile) may be used as an input for several other tasks. Hence we add a linear pipeline of fictitious null-size tasks that models the broadcast of the output to the target tasks. The DAG for the LU factorization of a $n \times n$ tiled matrix has $\frac{4}{3}n^3$ nodes, whereas the DAG for the Cholesky factorization has $\frac{2}{3}n^3$ nodes (and there are $O(n^2)$ fictitious tasks).

The running times of the linear kernels have been measured on the *mirage* platform, an heterogeneous system composed of two Intel hexacore processors X5650 at 2.67 GHz having 12 MB of L3 cache for a total of 12 cores and 36 GB of main memory, equipped with three NVIDIA Tesla M2070 GPUs having 6 GB of memory each. We associate the *blue* processors to the CPUs and the *red* processors to the GPUs. The running times were estimating by performing measurement with the MAGMA library [12] (using tiles of size 192×192 in double precision) and are given in Table I.

For communication costs, the average observed time to send one tile from a CPU to a GPU was approximatively 50 ms, thus all $C_{i,j}$ have been set to this value. The files sent and received by the tasks contain the value of the tiles. Since all tiles have the same size, we consider that for each edge (i, j) in the DAG, $F_{i,j} = 1$, one unit of memory corresponding to one tile.

Kernels	getrf	gemm	trsm_l	trsml_u	potrf	syrk
On CPUs	450	1450	990	830	450	990
On GPUs	10490	140	150	400	10490	150

Table I: Average running time in *ms* of the linear algebra kernels on a 192×192 tile

B. Results

1) SMALLRANDSET*:* To assess the absolute performance of the heuristics, we compare them to the optimal schedule found by the ILP described in Section IV. Note that SMALLRANDSET is the only set for which the ILP is able to compute a solution in a reasonable time. We aim at finding a schedule for each DAG in SMALLRANDSET with the smallest makespan as possible and under the same memory bound for each memory $M^{(blue)} = M^{(red)} = M^{(bound)}$.

First, we compute for each DAG $\mathcal{D}$ the makespan $Makespan_{HEFT}$ returned by the classical memory-oblivious HEFT algorithm and its maximum usage of each memory $M^{HEFT}_{\text{blue}}(\mathcal{D})$ and $M^{HEFT}_{\text{red}}(\mathcal{D})$. The idea is that the classical HEFT algorithm will not be able to schedule $\mathcal{D}$ on a platform with less than these amounts of *blue* and *red* memory. It is also clear that if the memory bounds respect $M^{(blue)} \geq M^{HEFT}_{\text{blue}}(\mathcal{D})$ and $M^{(red)} \geq M^{HEFT}_{\text{red}}(\mathcal{D})$, MEMHEFT will take exactly the same decisions as HEFT. Thus if $M^{(bound)} \geq max(M^{HEFT}_{\text{blue}}(\mathcal{D}), M^{HEFT}_{\text{red}}(\mathcal{D}))$, the performance of MEMHEFT will be the same as that of HEFT. Figure 10 reports the performances of MEMHEFT and MEMMINMIN if $M^{(bound)} = \alpha \times max(M^{HEFT}_{\text{blue}}(\mathcal{D}), M^{HEFT}_{\text{red}}(\mathcal{D}))$ with $\alpha \in [0, 1]$ being the relative memory compared to the amount needed by HEFT. Plain lines show the ratio of the average makespan of our heuristics, and of the solution returned by the ILP, over the makespan of HEFT. The average is computed over all DAGs successfully scheduled with the given memory bounds (to be read on the left scale). Dotted lines show the fraction of DAGs in SMALLRANDSET that our heuristics manage to schedule with the given memory bounds (to be read on the right scale).

We see that MEMHEFT and MEMMINMIN are really close to the optimal makespan when large amounts of memory are available. MEMMINMIN provides better results with a makespan overhead smaller than 50% w.r.t. HEFT, even when memory becomes critical. The dotted lines for MEMHEFT and MEMMINMIN in Figure 10 are indistinguishable, which means that both heuristics roughly fail on the same instances when memory becomes critical. MEMHEFT and MEMMINMIN both fail to provide a feasible schedule when the memory bounds is smaller to 35% of the amount required by HEFT. However, the ILP shows that there exists a feasible schedule for approximately 70% of the DAGs in SMALLRANDSET with this memory bound. Our heuristics can provide a feasible schedule for every DAG in SMALLRANDSET when the memory bound is greater than 75% of the amount required by HEFT, whereas, in theory, every DAG can be scheduled down to 60% of this amount. In addition to the global view for SMALLRANDSET, detailed results for the DAG of Figure 8 are provided in Figure 11.

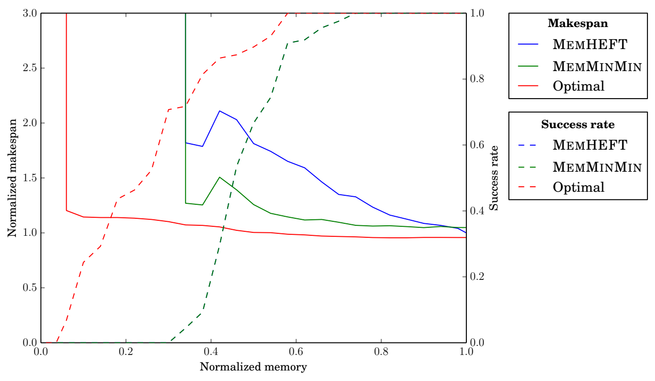

Figure 10: Results for SMALLRANDSET.

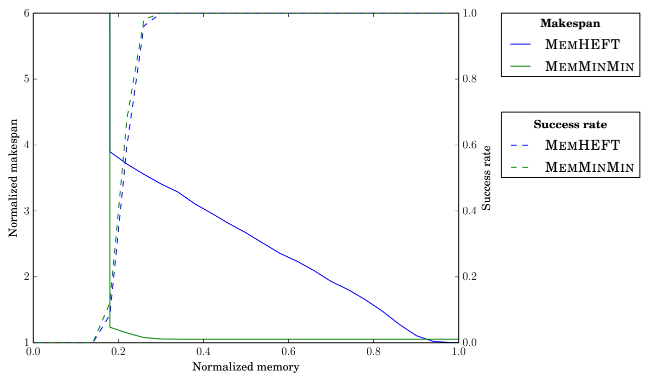

Figure 12: Results for LARGERANDSET.

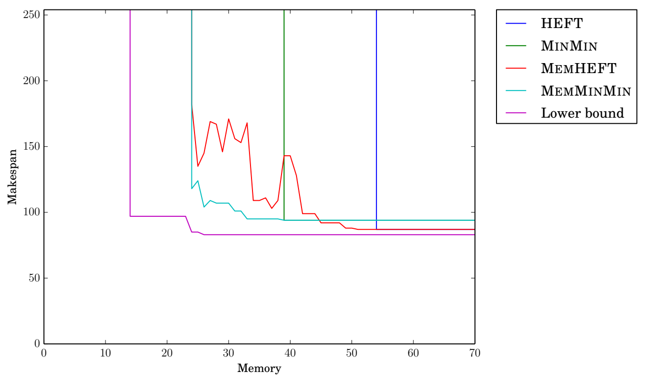

Figure 11: Makespan for the DAG in SMALLRANDSET depicted in Figure 8.

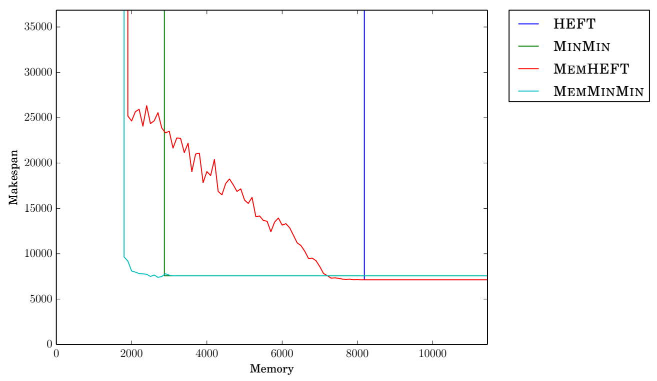

Figure 13: Makespan for the DAG in LARGERANDSET depicted in Figure 9.

2) LARGERANDSET: The same experimental procedure has been applied to LARGERANDSET, except that the optimal schedule cannot be computed in reasonable time anymore. The average relative makespan of our heuristics are depicted in Figure 12. We see that both MEMHEFT and MEMMINMIN succeed to schedule all the DAGs in LARGERANDSET with only 30% of the memory required by the classical HEFT algorithm. The average makespan of the schedules returned by MEMHEFT decreases almost linearly with the amount of available memory. Furthermore, for large amounts of memory, MEMHEFT provides slightly better results, while MEMMINMIN is clearly the best heuristic when memory is critical. MEMMINMIN provides only a 20% makespan overhead compared to HEFT while using 5 times less memory. Finally, specific results for the one DAG depicted in Figure 9 are provided in Figure 13.

3) LUSET and CHOLESKYSET: We provide results for numerical algebra sets corresponding to a 13×13 tiled matrix. Figure 14 depicts the results for LU factorization, whereas Figure 15 deals with Cholesky factorization. Contrarily to the previous section, MEMMINMIN seems to be the best heuristic when large amounts of memory are available. For both applications, MEMHEFT has a 10% makespan overhead compared to MEMMINMIN when large amounts of memory are available, but it requires far less memory to provide a feasible schedule. Indeed, Figure 14 shows that MEMMINMIN fails to schedule the LU factorization when each memory does not have enough space to store 155 tiles. However, MEMHEFT can still provide a feasible schedule with half available memory. This comes from the fact that in numerical algebra DAGs, a lot of non critical tasks are released early in the process and will eventually be immediately scheduled by MEMMINMIN, thereby filling up the memory. On the contrary, MEMHEFT will focus on the critical path of the DAG. Actually MEMHEFT fails when $M^{(bound)} \approx 85$ which approximately corresponds to the amount needed to store all the $13 \times 13 = 169$ tiles of the matrix on both memories. Since Cholesky factorization is performed on the lower half of the matrix (94 tiles), the results for the Cholesky factorization lead to similar conclusions.

Overall, both memory-aware heuristics achieve quite satisfactory trade-offs. In most cases, they are able to drastically reduce the amount of memory needed by HEFT or MINMIN, at the price of a relatively small increase in execution time. For small graphs, their absolute performance is close to the optimum as soon as half the memory required by HEFT is available.

VII. CONCLUSION

We have investigated the problem of scheduling a task graph on a dual-memory system, i.e. an heterogeneous platform made of two types of memories, with several processors attached to each memory. Dual-memory systems include

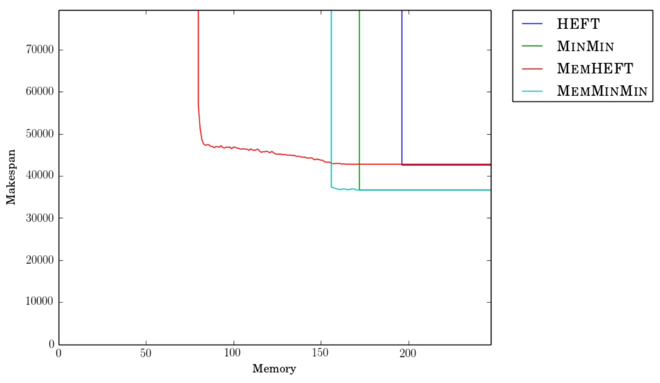

Figure 14: MemHEFT and MemMinMin results on the DAG representing an LU factorization of a 13×13 tiled matrix.

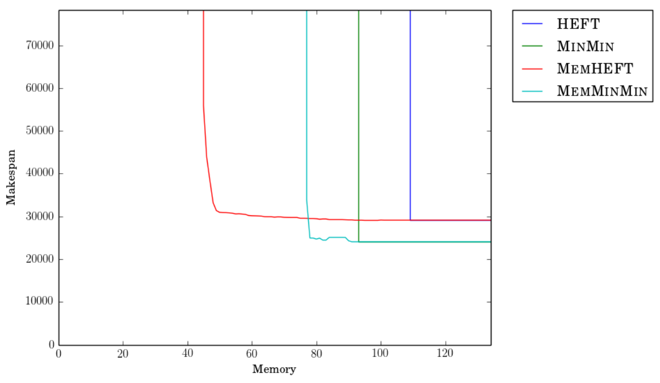

Figure 15: MemHEFT and MemMinMin results on the DAG representing an Choleski factorization of a 13×13 tiled matrix.

emerging hybrid computing platforms, which usually includes one or several accelerators (such as GPU) in addition to multicore CPUs. Our first contribution is to propose a simple model that captures the complexity of the problem. Given the NP-hardness of the problem (which follows from the complexity of the problem on trees, investigated in [1]), we have proposed several approaches. We first provide an exact resolution through the design of an intricate ILP which is able to compute an optimal schedule for medium-size instances (up to 30 tasks). Then, we propose two memory-aware heuristics for larger instances, which are the counterparts of the classical HEFT and MinMin algorithms. We have studied the performance of these new heuristics through extensive simulations on different task graphs, and compared them to the optimal solution for small instances.

An interesting future work would be to include some of the proposed heuristics in an actual runtime toolkit for hybrid platform such as StarPU [14]. It would also be of interest to adapt the heuristics to more complex platforms, such as hybrid platforms with several types of accelerators, and/or including more than two memories.

Acknowledgements

We thank Abdou Guermouche for providing the running times of the linear algebra kernels. This work was supported in part by the French Research Agency (ANR) through the Rescue and SOLHAR projects. Yves Robert is with Institut Universitaire de France.

References

[1] J. Herrmann, L. Marchal, and Y. Robert, “Model and complexity results for tree traversals on hybrid platforms,” in *Euro-Par 2013 - Parallel Processing*, ser. LNCS. Springer Verlag, 2013.

[2] H. Topcuoglu, S. Hariri, and M. Y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Trans. Parallel Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[3] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems,” *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, 2001.

[4] P. Chrétienne, E. G. Coffman Jr., J. K. Lenstra, and Z. Liu, Eds., *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[5] M. A. Palis, J.-C. Liou, and D. S. L. Wei, “Task clustering and scheduling for distributed memory parallel architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 1, pp. 46–55, 1996.

[6] I. Ahmad and Y.-K. Kwok, “On exploiting task duplication in parallel program scheduling,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 9, pp. 872–892, 1998.

[7] I. T. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler, “End-to-end quality of service for high-end applications,” *Computer Communications*, vol. 27, no. 14, pp. 1375–1388, 2004.

[8] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, “Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR,” *Int. Journal of High Performance Computing and Applications*, 2008.

[9] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, “Scheduling data-intensiveworkflows onto storage-constrained distributed resources,” in *CCGRID’07*. IEEE, 2007.

[10] J. W. H. Liu, “An application of generalized tree pebbling to sparse matrix factorization,” *SIAM J. Algebraic Discrete Methods*, vol. 8, no. 3, 1987.

[11] ——, “On the storage requirement in the out-of-core multifrontal method for sparse factorization,” *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 249–264, 1986.

[12] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, “Numerical linear algebra on emerging architectures: The plasma and magma projects,” *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012037, 2009.

[13] M. Horton, S. Tomov, and J. Dongarra, “A class of hybrid lapack algorithms for multicore and gpu architectures,” in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, july 2011, pp. 150 –158.

[14] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, “Starpu: a unified platform for task scheduling on heterogeneous multicore architectures,” *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[15] S. Venugopalan and O. Sinnen, “Optimal linear programming solutions for multiprocessor scheduling with communication delays,” in *ICA3PP (1)*, 2012, pp. 129–138.

[16] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović, “Towards the optimal solution of the multiprocessor scheduling problem with communication delays,” in *In MISTA Proceedings*, 2007.

[17] T. N’Takpé, F. Suter, and H. Casanova, “A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms,” in *ISPDC*, 2007, pp. 250–257.

[18] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, “A class of parallel tiled linear algebra algorithms for multicore architectures,” *Parallel Computing*, vol. 35, no. 1, 2009.