SPAGHETtI: Scheduling/Placement Approach for Task-Graphs on HETerogeneous archItecture

Denis Barthou[1,2] and Emmanuel Jeannot[2]

[1] Bordeaux Institute of Technology, France
[2] Inria, LaBRI, France

Abstract. We propose a new algorithm, called SPAGHETtI, for static scheduling tasks on an unbounded heterogeneous resources where resources belongs to different architecture (e.g. CPU or GPU). We show that this algorithm is optimal in complexity $O(|E||A|^2 + |V||A|)$, where $|E|$ is the number of edges, $|V|$ the number of vertices of the scheduled DAG and $|A|$ the number of architectures – usually a small value – and that it is able to compute the optimal makespan. Moreover, the number of resources to be used for executing the schedule is given by a linear time algorithm. When the resources are bounded we provide a method to reduce the number of necessary resources up to the bound providing a set of compromises between the makespan and the size of the infrastructure.

1 Introduction

Directed acyclic graphs (DAGs) have been used to model [7,8,15], execute [2,5,12] and predict [14] the performance of parallel applications. There exists many scheduling algorithms for mapping tasks of a DAG onto the resources of parallel machines [13,17,20]. A lot of work have been proposed to schedule task graphs on heterogeneous resources when execution and communication time depend on the machine that executes a task [3,16,17]. However, recent advances in High-Performance Computing (HPC) have led to two important considerations:

- HPC systems feature a relatively low heterogeneity. Contrary to proposed solutions of the literature where each individual machine can perform differently, one often face a fix number of architectures (e.g. CPU, GPU, MIC, etc.) where performance is homogeneous.
- HPC systems and their applications are of very large-scale. Top end HPC systems can have as many as hundreds of thousands of processors. The tiled version of the dense Cholesky factorization for instance has more than 10 million tasks (matrix of order 204800 and tiles size of 512). Therefore, the complexity to schedule the DAGs is crucial in this setting.

In this paper, we propose a new static scheduling algorithm designed for this kind of systems. Instead of considering each individual processor independently it considers the architectures of the target machine. Within each architecture the communication and execution time is considered homogeneous. Thanks to

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 174–185, 2014.

that feature, for an unbounded number of resources, it is able to schedule the input graph optimally in terms of makespan, with an optimal complexity of $O(|E||A|^2 + |V||A|)$ where $|E|$ is the number of edges, $|V|$ of vertices of the DAG and $|A|$ the number of architectures and potentially resorting to task duplication.

The remaining of the paper is organized as follows. In Section 2, we discuss the related work. The models are described in Section 3. The algorithm is detailed in Section 4. How to go from an unbounded number of resources to a bounded number is discussed in Section 5. Experimental results are provided in Section 6.

2 Related Work

Static scheduling task graphs on homogeneous resources is NP-hard even for two machines (reduction from 2-partition [9]). However, for unbounded resources and no communication cost it is clearly in P as it requires to use new resources (resource augmentation) to have a schedule of the length of the critical path. In the case of communication cost, optimal scheduling can be found for some special input graph only (without [19] or with [1] duplication).

There exists a lot of static scheduling heuristics for heterogeneous scheduling (see [6] for some examples. If duplication is not allowed, HEFT [17] provides a good schedule in a reasonable complexity $O(|V|^2p)$, with p the number of processing units. In [6], it has been experimentally shown that HEFT is one of the best heuristics (in terms of makespan) for random graphs among 20 different heuristics. In case duplication is allowed, TANH [3] is a heuristic of interest for our study as it provides a low complexity $(O(|V|(p \log p) + |V|^2))$ and is optimal under some hypothesis. The authors show that TANH provides an optimal schedule (in terms of makespan) if a "*A fork node i that is not a join node is assumed to have the same execution time on all processors.*". Such hypothesis does not hold in many cases. For instance in the Cholesky task graph, the POTRF task is a fork task that is not a join task and its runtime is very different if you execute it on a CPU or on a GPU.

In conclusion static scheduling heuristics have a complexity that depends on the number of processors and are not able in the general case, due to NP-completeness, to provide an optimal schedule.

3 Models and Definitions

We consider an application modeled by a directed acyclic task graph (DAG) $G = (V, E)$ where V is the set of tasks to be executed and E represents precedence constraints between tasks. The execution model of the DAG is close to the macro-dataflow model where a task can be executed only after all its predecessors have terminated and when communications from its predecessors and this task have been performed. However, it differs in the way costs are modeled.

We want to model a large platform where we have different architectures. Think for instance of a node with a set of multicore processors (a NUMA machine with several hundreds of cores) with some accelerators (e.g. GPU cards having each several hundreds of CUDA cores or Xeons Phi each featuring 60 cores with 4 threads each). In this case, we assume that the communication cost between two architectures is the same whatever the actual instances that are sending and receiving the data. Moreover, the communication costs are considered constant when data move within one architecture (whether this task is executed on the same instance as its predecessor or not). This later assumption is different from the standard DAG scheduling model where a distinction is made if the communication is occuring within the same instance (has no cost) or between different instances (has a non zero cost). This is justified as follows. First, in our model, this constant cost can be zero in order to neglect intra-architecture communications compared to inter-architecture communications. Second, we want to produce a schedule where high-level decisions are taken such has: "*On which architecture should I schedule this task* ?" We think the impact of locality on a multi-architecture machine is more important than locality inside one given homogeneous architecture. Third, the assumption of constant communication time within an architecture makes all the difference theoretically speaking: it is this assumption that allows us to find an optimal solution. Finally, the experiment section will show that it leads to predictable execution time and is therefore reasonable in real settings.

Formally, let ω be the communication time function, defined for each edge of the graph, and τ is the execution time function, defined for each vertex. We consider a set of different architectures, A. The communication time for a given edge depends on the architecture executing the vertices of this edge. Hence, the function ω is defined over $E \times A^2 \to \mathbb{R}$: For each edge, we have a communication matrix of order $|A| * |A|$ that provides the communication times of this edge depending on the source and destination architecture. Similarly the execution time function is defined as $V \times A \to \mathbb{R}$: for each task we have a vector of execution time of order $|A|$ (see an example in Fig. 1).

Definition 1 (Start time). *For a task graph G and an architecture set A, the start time is a function:*

$$\begin{aligned} \theta : V \times A &\to \mathbb{N} \\ i, j &\to t \end{aligned}$$

that associates to task i and architecture j a time t for i to start on j. We denote the starting time of task i on architecture j: $\theta_i[j]$.

The start time is a total function, defined for all vertices and architectures. It does not imply that tasks are systematically duplicated on all architectures, but only represents possible starting times according to architectures. The earliest completion time is defined as the minimal time to start a task, added to the time to execute the task, considering all possible architectures: $C_i^{earliest} = \min_{j \in A} \theta_i[j] + \tau_i[j]$ The makespan is then simply deducted: $C_{\max} = \max_{i \in V} C_i^{earliest}$. The makespan usually involves the latest completion time, when tasks are duplicated. The earliest completion time is equal to the latest completion time for non-duplicated tasks. This is not a limitation, since it is always possible to define an additional sink task, having as predecessors the tasks initially with no successors. Besides, our mapping algorithm will ensure that the tasks with an earliest completion time equal to the makespan are not duplicated.

The mapping function defines more precisely the resource executing task i:

Definition 2 (Mapping). *A mapping of a task graph G is a function*

$$\begin{aligned} \mu : V \times A &\to \mathbb{N} \cup \{\perp\} \\ i, j &\to r \end{aligned}$$

that associates to each task i and architecture j the resource number r that executes i. When i is not executed on architecture j, r corresponds to the special value $\perp$. When the same task is mapped to different architectures, there is duplication. We denote the resource executing task i on architecture j: $\mu_i[j]$.

Definition 3 (Constraints). *Given a graph G, a set of architectures A and a vector of resources $r = (r_k)_{k \in A}$, the functions μ and θ define resp. a valid mapping and schedule if and only if the following constraints are checked:*

1. *Resource constraint. One task is executed at a time on the same resource.*
$$\begin{aligned} \forall i, j \in V, k \in A, i \neq j, \mu_i[k] = \mu_j[k] \neq \perp \Rightarrow (\theta_i[k] + \tau_i[k] \leq \theta_j[k]) \\ \vee (\theta_j[k] + \tau_j[k] \leq \theta_i[k]) \end{aligned} \quad (1)$$
2. *Architecture constraint. Resources are bounded by r:*
$$\forall i \in V, k \in A, \mu_i[k] \leq r_k \quad (2)$$
3. *Dependence constraints. The start time follows the precedence constraint and communication costs:*
$$\forall (i, j) \in E, \forall k, \exists h, \theta_j[k] \geq \theta_i[h] + \tau_i[h] + \omega_{ij}[h, k] \quad (3)$$

4 The SPAGHETtI Algorithm

We consider here the computation of the minimum makespan when there is no resource constraint (1) and no architecture constraint (2). Within this formulation, it is possible to define a schedule and a mapping function giving for each task the architecture(s) where it executes.

4.1 Minimizing Makespan

Consider first the case where there is only one architecture available, i.e. $|A| = 1$. Then ω and τ are only functions of tasks. The optimal makespan can be evaluated by computing the earliest start time of each task. According to the dependence constraint (3), this start time fulfills the following property:

$$\theta_j^{earliest} = \max_{(i,j) \in E} (\theta_i^{earliest} + \tau_i + \omega_{ij})$$

We can arbitrarily define the earliest start time for tasks with no predecessor in G as 0. This formulation then corresponds to a longest path problem on the DAG G (critical path). This can be solved in $\mathcal{O}(|V|+|E|)$ time with a topological sort and then the evaluation in topological order of the function $\theta^{earliest}$.

Now, consider the case where $|A| \geq 1$. The dependence constraint defines the value of the earliest start time as:

$$\theta_j^{earliest}[k] = \max_{(i,j)\in E} \min_{h\in A}(\theta_i^{earliest}[h] + \tau_i[h] + \omega_{ij}[h,k]).$$

Using $(\min, +)$ notation algebra, where the addition corresponds to a min and multiplication to $+$, the min term can be rewritten into: $\sum_{h\in A}(\theta_i^{earliest}[h] * \tau_i[h] * \omega_{ij}[h,k]$. This corresponds to a matrix vector product with θ_i and τ_i vectors indexed by A and $\omega_{i,j}$ a square matrix of rank $|A|$. The vector definition of $\theta_j^{earliest}$ is therefore:

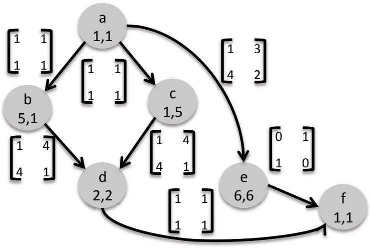
$$\theta_j^{earliest} = \max_{(i,j)\in E} \theta_i^{earliest} * \mathrm{diag}(\tau_i) * \omega_{i,j} \tag{4}$$

with max the component-wise maximum and $\mathrm{diag}(\tau_i)$ the diagonal matrix obtained from the vector. This recursive definition of $\theta_j^{earliest}$ is similar to the case where $|A| = 1$, and leads to the definition of the SPAGHETtI algorithm.

Algorithm 1. Compute the earliest starting time for each vertex in G

```
Input: $G = (V, E)$ // The input DAG
Input: $\tau$ // Function defining the duration time vector
Input: $\omega$ // Function defining the communication time vector
forall the $i \in G$ do // Assign a time vector, for all architectures
    $\theta_i \leftarrow 0$
$C_{\max} \leftarrow 0$
$S \leftarrow$ Topological_sort($G$)
forall the $i \in S$ do // Visit $G$ in topological order, starting with source
    for every vertex $j$ predecessor of $i$ in $G$ do
        $\theta_i = \max(\theta_i, \theta_j * \mathrm{diag}(\tau_j) * \omega_{j,i})$// Element-wise maximum on vectors
    $C_{\max} \leftarrow \max(C_{\max}, \max_{k\in A} \theta_i[k] + \tau_i[k])$
```

Figure 1 shows an example of the schedule and makespan computed by SPAGHETtI on a graph, for two architectures, CPU and GPU. CPU values are put in the first row/column of vectors and matrices, GPU in the second. For instance, CPU→CPU communication between a and e takes 1, CPU→GPU takes 3. The earliest starting time for task a is 0 for both architectures. The starting time for task b, when started on CPU, is at least the time to complete a on CPU and then communicate with b, or complete a on GPU and communicate accross architectures. This leads to a starting time of 2. This is the same case for GPU, and for task c. Task e on CPU cannot complete before either task a has completed on CPU and CPU→CPU communication has finished (duration 1), or task a has completed on GPU and GPU→CPU communication has finished (duration 4): the earliest starting time for e on CPU is therefore 2. We let the reader continue the reasoning and check the values of the table on the right.



Task	$\theta^{earliest}_{CPU}$	$\theta^{earliest}_{GPU}$	Mapping
a	0	0	CPU
b	2	2	GPU
c	2	2	CPU
d	7	7	CPU
e	2	3	CPU
f	8	9	CPU

Fig. 1. On the left, the task graph with the values of τ for each task and ω for each edge. On the right, the earliest starting time for each task, given when the task starts on CPU and on GPU. Then the last column corresponds to the architecture where the task is mapped in order to reach the optimal makespan (9 in this example).

Theorem 1. *Algorithm 1 computes the optimal makespan of* G *with the optimal complexity* $\mathcal{O}(|A|^2 * |E| + |A| * |V|)$.

Proof. First, let us show that the SPAGHETtI algorithm computes indeed the optimal makespan. Assume that Algorithm 1 does not compute the optimal makespan. There exists a scheduling function θ' verifying the dependence constraint (3) such that for all architectures k, $\theta'_{sink}[k] \leq \theta_{sink}[k]$ and for at least one architecture, this inequality is strict. Such relation is denoted $\theta'_{sink} < \theta_{sink}$. Consider a task i_0, minimal according to the topological order, such that $\theta'_i < \theta_i$. θ_{i_0} is defined as:

$$\theta_{i_0} = \max_{(j,i_0)\in E}(\theta_j * \mathrm{diag}(\tau_j) * \omega_{j,i_0}).$$

As $\theta_j = \theta'_j$ for all the predecessors j of i_0 and there exists a $k \in A$ such that $\theta'_{i_0}[k] < \theta_{i_0}[k]$, we have:

$$\theta'_{i_0}[k] < \max_{(j,i_0)\in E}\min_{h\in A}(\theta'_j[h] * \tau_j[h] * \omega_{j,i_0}[h,k]).$$

Thus there exists a predecessor j of i_0 such that for all architecture $h \in A$:

$$\theta'_{i_0}[k] < \theta'_j[h] * \tau_j[h] * \omega_{j,i_0}[h,k].$$

This is in contradiction with the dependence constraint (3), and contradicts the definition of θ'. Hence Algorithm 1 computes the optimal makespan.

Now line 7 corresponds to $O(|A|^2 * |E|)$ operations, the $|A|^2$ term coming from the matrix vector product $\theta_j * \mathrm{diag}(\tau_j) * \omega_{j,i}$. Line 8 takes $O(|A| * |V|)$ operations due to the max operation. The total complexity corresponds to the size of the inputs. Since the makespan may depend on all of them, this shows the complexity is optimal. □

4.2 Mapping Tasks to Architectures

Finding a mapping function corresponds to finding one or several architectures for each task, compatible with earliest starting time constraints. As there is no resource constraints, μ is here an indicator function returning a boolean: a task i is mapped on an architecture $j \in A$ if $\mu_i[j] = 1$ otherwise $\mu_i[j] = 0$. A task is duplicated on two different architectures $j, k, j \neq k$ if $\mu_i[j] = \mu_i[k] = 1$.

For all tasks with no successor in G, the architecture is chosen so that the earliest completion time can be attained:

$$\forall h \in A, h = \min\{k \in A \mid \theta_i[k] + \tau_i[k] = C_i^{earliest}\} \Rightarrow \mu_i[h] = 1. \qquad (5)$$

Note that these tasks are not duplicated, since h is uniquely defined. The makespan corresponds to the earliest completion time of one of these tasks, hence the mapping here is chosen so that the optimal makespan is reached.

For all the other tasks, the dependence constraint guides the choice of architecture that can execute them: Consider a task $i \in G$ and an edge $(i, j) \in E$. Assume j is mapped on architecture $k \in A$, then the schedule computed by the SPAGHETtI algorithm ensures there exists an architecture $h \in A$ such that $\theta_i[h] \leq \theta_j[k] - \tau_i[h] - \omega_{ij}[h, k]$. This defines a value for μ_i:

$$\forall (i, j) \in E, \forall k, l \in A,$$
$$\mu_j[k] = 1 \wedge l = \min\{h \in A \mid \theta_i[h] \leq \theta_j[k] - \tau_i[h] - \omega_{ij}[h, k]\} \Rightarrow \mu_i[l] = 1. \quad (6)$$

An alternative definition of μ can prevent useless task duplication, whenever possible. Instead of Equation (6), the following equation can be used:

$$H_i = \{h \mid \forall (i, j) \in E, \forall k \in A, \mu_j[k] = 1 \Rightarrow \theta_i[h] \leq \theta_j[k] - \tau_i[h] - \omega_{ij}[h, k]\},$$
$$H_i \neq \emptyset \Rightarrow \mu_i[\min H_i] = 1. (7)$$

When this equation does not define a value for μ_i, Equation (6) has to be used and duplication is necessary. Equations (5), (6) and (7) define recursively the function μ: Starting from tasks with no successor, μ is defined for all tasks in a reverse topological order. The definition of μ shows that this computation requires $\mathcal{O}(|A|^2|E|)$ operations when applying definitions (6) or (7) and $\mathcal{O}(|A||V|)$ operations when applying definition (5). This is the optimal complexity since, as for the schedule, it corresponds to the size of the inputs G, τ and ω. Therefore, this procedure, combined with the SPAGHETtI algorithms provides a solution that is optimal in terms of makespan and complexity.

Figure 1 shows the result of the mapping computation on the task graph. As the task f as a lower completion time $8 + 1 = 9$ when executed on CPU, this is the mapping of this task. Task e and d are indifferently mapped to CPU or GPU (here CPU, ordered first). For task b, there is only one possible mapping to ensure that d is scheduled at time 7: b has to be scheduled on GPU.

4.3 Determining the Number of Resources for Each Architecture

The required amount of resources for each architecture is not given by the previous algorithms. To determine this number of resources we use a greedy algorithm that allocates task to architecture instances, extending the previous architecture mapping computed in the previous section and computing the actual instance $\mu_i[k]$ of task i when mapped on architecture k. For each architecture we consider the tasks by increasing start time and we allocate them to the first resource of the architecture that can respect the start scheduling constraints. If no resource is available we proceed with resource augmentation and create a new instance of this architecture. Therefore, the number of resources used is the minimal number of resources that respect the schedule (i.e. the task start time). Moreover, this allocation is optimal in terms of platform dimensioning only if there is no sufficient slack in the schedule to delay tasks in order to save resources.

5 Exploring Tradeoffs for Heterogeneous Machines

Here, we deal with the case where the number of resources is higher than the available ones. There exists several ways of reducing the number of resources used by a schedule. In homogeneous setting an effective way was explored by the Pyrros project [19] where, after DSC [20] clusters were merged using the *work profiling method* of [10]. Another technique, presented in the context of register allocation, consists in adding some dependence edges in the graph in order to reduce the number of simultaneously live variables [18].

In heterogeneous environments, merging architectures has no meaning. We propose here a method similar to the one proposed for register allocation, where instructions are replaced by tasks and resources are processing units instead of registers. We reduce the inherent parallelism of the task graph by iteratively adding edges and then re-computing the schedule, the mapping and the number of resources until we reach the target number of resources. The procedure is depicted in Algorithm 2.

Algorithm 2. Adding n edges to the DAG G to reduce its parallelism

```
Input: $G = (V, E)$ // The input DAG
Input: $n$ // Number of edges to add
$S \leftarrow$ Topological_sort($G$)
$I \leftarrow$ Interference_graph($G$);
forall the $n$ edges to be added do // We will add $n$ edges
    $i \leftarrow$ Highest_degree_node($I$)
    $j \leftarrow$ Highest_degree_node(neighbor($i$))
    if $i \prec_S j$ then // If $i$ is before $j$ in the topological order
        Add $(i, j)$ in $G$ // Communication cost is set to 0
    else
        Add $(j, i)$ in $G$ // Communication cost is set to 0
    Remove $(i, j)$ in $I$ // and decrease degree of $i$ and $j$
```

To add edges to the graph in order to reduce its parallelism, we first sort nodes in topological order. Then, we build the *interference graph* I of the DAG. In the interference graph, vertices are the same as in the original DAG. There is an edge between two vertices if there is no path between them in the DAG (they could be scheduled in parallel). In this interference graph we choose the node i of highest degree and a neighbor of i of highest degree. Then, this edge is added to the DAG G and the interference graph is updated. We iterate until

n edges have been added. Therefore we add a batch of n edges before applying again the SPAGHETtI algorithm. The rational behind adding several edges at the same time is to amortize the interference graph construction. The rational behind choosing the highest degree nodes in the interference graph is that a node with high degree has a lot of freedom in terms of parallelism and we are therefore more likely to impact the whole graph parallelism by reducing the parallelism of this kind of nodes. We avoid adding cycles in the original DAG: the added edge is directed so that it respects the topological order (line 6).

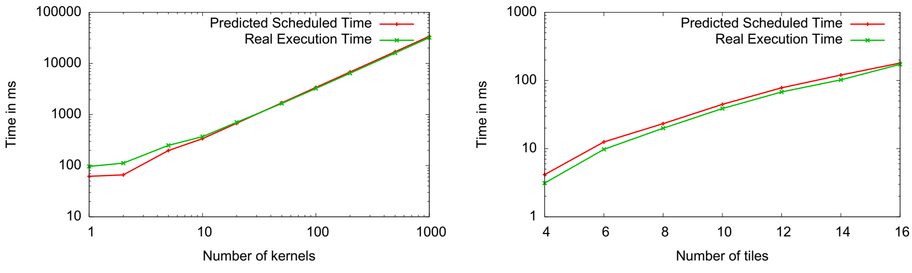
Moreover, each time we add a set of n edges, we compute the makespan of the new SPAGHETtI's schedule. This outputs a new compromise between the execution time and the number of resources. Hence, with this procedure we explore a full set of compromises (time vs. resources) until we reach the required bound. This is helpful for decision makers to correctly dimension their platform. In the following experiments, n was chosen between 10 and 100.

6 Experimental Results

We have implemented all the algorithms and procedures of the previous section. They take an input DAG, the communication and computation cost of each task on each architecture and the target number of resources for each architecture. In the following experiments intra-architecture communications are always zero.

We have also designed a simple runtime system that executes the static schedule on the given environments. In our experiments we have used nodes featuring 2 6-cores intel Xeons (X5650) at 2.67GHz with 36 Gb of RAM and 3 NVIDIA Tesla M2070 GPU at 1.15 GHz with 6 Gb of memory.

We have coded the dense tiled Cholesky factorization [11]. It features 4 kernels (POTRF, TRSM, SYRK and GEMM) that are executed using the Intel MKL library 12.1.9 for the CPUs and the CUBLAS version 4.2 for the GPUs. The Cholesky DAG can be seen here [4].



(a)

(b)

Fig. 2. Model validation experiments on (a) a chain of SYRK kernels alternatively on CPU and GPU, (b) on a tiled Cholesky factorization with 4096x4096 tile size

Model Validation. To validate our model we have executed a real schedule, measured the execution time of each kernel and compared the predicted schedule time with the measured values.

In Fig. 2(a), we execute a chain of SYRK kernels that are scheduled alternatively between a GPU and CPU. We see that as the chain size increases, the performance between the predicted time and the actual execution time becomes closer. This validates our execution and inter-architecture communication model.

In Fig. 2(b), we execute the Cholesky factorization[1] using tile size of 4096 and the decomposition of the matrix varies between 4 and 16 tiles (hence, the order of the matrix varies between 16384 and 65536). Here, we see that the predicted execution time is just a little higher than the real execution time. This validates the kernel execution time and communication time within a GPU as all the tasks, in this case, are scheduled on the GPUs.

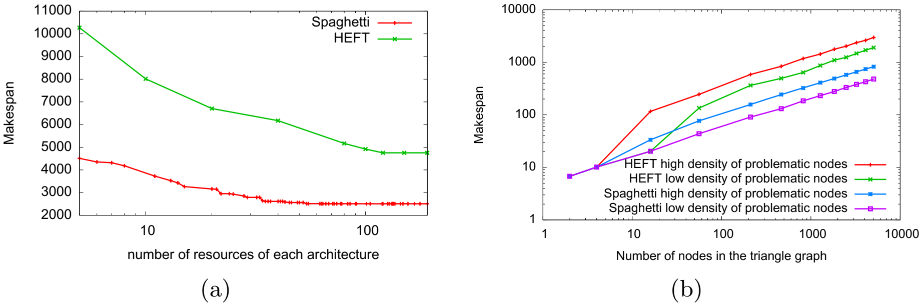


Fig. 3. HEFT and SPAGHETtI comparison (a) for bounded number of resources for the Cholesky Graph, (b) for unbounded number of resources in case of duplication

Comparison with HEFT. Being a list scheduling algorithm, HEFT is not able to make a short-term sacrifice to achieve a gain in the long term. This is exemplified with Fig. 3(a). In this Figure, we schedule a Cholesky DAG of 1540 nodes on two different architectures. Communicating within an architecture is free but communicating between architecture is very costly. In this case, the first task to be scheduled is faster on architecture 1 than on architecture 2 and the other tasks are faster on architecture 2. HEFT will execute the first task on architecture 1 and stay on this architecture until the end of the execution. On the other hand, the execution cost of the tasks on architecture 2 can amortize the communication time: SPAGHETtI pays the cost of executing the first task on architecture 2 and continues to execute all the tasks on this architecture. The optimal makespan is 2505 for 191 resources of architecture 2. We output all the compromises found by our method between 191 and 5 resources. For 191 resources, SPAGHETtI's makespan is 1.9 faster than the HEFT one. But if we reduce the number of resources to 5 for both architectures, SPAGHETtI still

[1] The factorization was checked correct by post-processing the result.

outperforms HEFT by a factor of 2.2. We explain the increase of performance ratio as follows. For a large number of resources SPAGHETtI does not need to use duplication but when the number of resources decreases, SPAGHETtI finds that duplication reduces the makespan even more. Indeed, it starts using this feature when the number of available resources is lower than 105.

In order to assess the importance of duplication, we have tested the case of a *triangle DAG*, where from time to time, two nodes (called *problematic nodes*) sharing the same predecessors, have an opposite behavior in terms of execution time (one is more efficient on one architecture while the other is more efficient on the other architecture) and in terms of communication time (going from the architecture they favor to the other architecture is very costly). The other tasks are homogeneous (they have the same execution time on every architectures). In this case, it is better to duplicate nodes that are predecessors of these problematic nodes in order to avoid to pay the communication cost while these nodes are executed on their privileged architecture. This is what is depicted on Fig 3(b) where we see that, the inability of HEFT to duplicate nodes, adds a big overhead in the makespan. We also see that for small number of nodes, HEFT and SPAGHETtI perform identically: this is due to the fact that there is no problematic nodes for small instances.

7 Conclusion

Being able to schedule a DAGs on a large parallel machine is a challenge. Most heterogeneous static scheduling heuristics have a complexity that depends on the number of resources. In this paper we propose to classify the resources by architecture in order to reduce the complexity of the scheduling process and to cope with modern HPC environments where the heterogeneity is relatively low. We also use a model where the communication time depends only on the source and destination architecture and not on the instances of these architecture. Thanks to this hypothesis, we are able to provide an optimal mapping strategy with a very low complexity. We then show that we can find the minimal number of resources required to respect the schedule start time and we are able to propose a set of compromises (makespan vs. platform size) in order to help decision makers to dimension their environment depending on the time-to-solution constraint they impose. Results show that the proposed model is verified in some real settings and that we are able to amortize the execution of some task on suboptimal resources or to duplicate tasks when necessary.

Future works are directed towards a better optimization of the part where we switch from unbounded to bounded resources. We plan to do this by exploiting the slack of the schedule and map tasks on suboptimal resources as long as the schedule length is not increased.

Acknowledgement. We would like to thank Valentin Fréchaud for his help in the implementation and test of the SPAGHETtI method.

References

1. Ahmad, I., Kwok, Y.K.: On exploiting task duplication in parallel program scheduling. IEEE Transactions on Parallel and Distributed Systems 9(9), 872–892 (1998)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23(2), 187–198 (2011)
3. Bajaj, R., Agrawal, D.P.: Improving scheduling of tasks in a heterogeneous environment. IEEE Transactions on Parallel and Distributed Systems 15(2), 107–118 (2004)
4. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: Dague: A generic distributed dag engine for high performance computing, innovative computing laboratory technical report. Tech. rep., ICL-UT-10-01 (2010)
5. Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., Dongarra, J.: Dague: A generic distributed dag engine for high performance computing. Parallel Computing 38(1), 37–51 (2012)
6. Canon, L.-C., Jeannot, E., Sakellariou, R., Zheng, W.: Comparative evaluation of the robustness of dag scheduling heuristics. In: Grid Computing, pp. 73–84. Springer (2008)
7. Chong, F.T., Sharma, S.D., Brewer, E.A., Saltz, J.: Multiprocessor runtime support for fine-grained, irregular dags. Parallel Processing Letters 5(04), 671–683 (1995)
8. El-Rewini, H., Lewis, T., Ali, H.: Task Scheduling in Parallel and Distributed Systems. Prentice Hall (1994)
9. Garey, M., Johnson, D.: A Guide to the Theory of NP-Completeness. W.H. Freeman and company, New York (1979)
10. George, A., Heath, M.T., Liu, J.: Parallel cholesky factorization on a shared-memory multiprocessor. Linear Algebra and its applications 77, 165–187 (1986)
11. Gustavson, F.G., Karlsson, L., Kågström, B.: Distributed sbp cholesky factorization algorithms with near-optimal scheduling. ACM Transactions on Mathematical Software (TOMS) 36(2), 11 (2009)
12. Jeannot, E.: Automatic multithreaded parallel program generation for message passing multiprocessors using parameterized task graphs. In: International Conference on Parallel Computing (2001)
13. Leung, J.Y.T. (ed.): Handbook of Scheduling. Chapman & Hall/CCR (2004)
14. Mak, V.W., Lundstrom, S.F.: Predicting performance of parallel computations. IEEE Transactions on Parallel and Distributed Systems 1(3), 257–270 (1990)
15. Sinnen, O.: Task scheduling for parallel systems, vol. 60. Wiley. com (2007)
16. Tang, X., Li, K., Liao, G., Li, R.: List scheduling with duplication for heterogeneous computing systems. J. of Parallel and Distributed Computing 70(4), 323–329 (2010)
17. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
18. Touati, S.-A.-A., Eisenbeis, C.: Early control of register pressure for software pipelined loops. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 17–32. Springer, Heidelberg (2003)
19. Yang, T., Gerasoulis, A.: Pyrros: Static Task Scheduling and Code Generation for Message Passing Multiprocessor. In: Supercomputing 1992, pp. 428–437. ACM, Washington D.C (1992)
20. Yang, T., Gerasoulis, A.: DSC Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE Trans. on Parallel and Distributed Systems 5(9) (1994)