Low-Cost Tuning of Two-Step Algorithms for Scheduling Mixed-Parallel Applications onto Homogeneous Clusters

Sascha Hunold[‡]
International Computer Science Institute
Berkeley, CA, USA
Email: sascha@icsi.berkeley.edu

***Abstract*—Due to the strong increase of processing units available to the end user, expressing parallelism of an algorithm is a major challenge for many researchers. Parallel applications are often expressed using a task-parallel model (task graphs), in which tasks can be executed concurrently unless they share a dependency. If these tasks can also be executed in a data-parallel fashion, e.g., by using MPI or OpenMP, then we call it a mixed-parallel programming model. Mixed-parallel applications are often modeled as directed acyclic graphs (DAGs), where nodes represent the tasks and edges represent data dependencies. To execute a mixed-parallel application efficiently, a good scheduling strategy is required to map the tasks to the available processors.**

Several algorithms for the scheduling of mixed-parallel applications onto a homogeneous cluster have been proposed. MCPA (Modified CPA) has been shown to lead to efficient schedules. In the allocation phase, MCPA considers the total number of processors allocated to all potentially concurrently running tasks as well as the number of processors in the cluster. In this article, it is shown how MCPA can be extended to obtain a more balanced workload in situations where concurrently running tasks differ significantly in the number of operations. We also show how the allocation procedure can be tuned in order to deal not only with regular DAGs (FFT), but also with irregular ones. We also investigate the question whether additional optimizations of the mapping procedure, such as packing of allocations or backfilling, can reduce the makespan of the schedules.

I. Introduction

Expressing the potential parallelism of an algorithm in the field of scientific programming is a challenging problem. There are numerous ways to enable applications to exploit different processing units at the same time. Traditionally, two main models of parallel programming have emerged, the *data-parallel* model and the *task-parallel* model. Even though there is not always a sharp border between them, in the data-parallel model the same operation is applied in parallel to different data (SPMD), whereas in the task-parallel model different operations can be applied to the data (MPMD). Often, concurrently running tasks are executed by one processor, as in a thread model, e.g., Intel's TBB. In the world of distributed memory machines, e.g., clusters or grids, a data-parallel model is often associated with MPI, whereas executing tasks on the grid would be considered task-parallel, e.g., by using Condor's DAGMan. From a software engineering perspective reusing modules and tasks is essential for a rapid and robust development of applications. For this reason, workflows represent an interesting class of applications to be executed in parallel environments such as cluster systems.

In a workflow, nodes denote the tasks and the edges denote the dependencies between the tasks. If tasks of a workflow can also be executed by multiple processors, the resulting parallel model is called mixed-parallel. Thus, mixed-parallel applications (workflows) can be modeled as directed acyclic graphs (DAGs) of data-parallel tasks. Such structures arises often in scientific applications [1]. One can also think of a coordination program for scientific programming, e.g., a MATLAB program calling routines that are implement using a parallel back-end [2], [3].

It is now the challenge to schedule the DAG of tasks to the parallel platform by minimizing the resulting execution time of the entire workflow. In this work, all parallel tasks are *moldable*, i.e., a task can be executed by any number of processors. Thus, a scheduling algorithm has to determine the number of processors that are assigned to each task.

In this article, we focus on the scheduling of mixed-parallel applications onto homogeneous clusters, i.e., all processors are identical and are interconnected by a single high performance network. In this context, the moldable tasks are called multi-processor tasks (M-tasks). Several algorithms for scheduling mixed-parallel application onto a homogeneous cluster have been proposed. Many of these approaches have a fairly high computational complexity, which questions their practical applicability. Due to these limitations, algorithms with a lower complexity have been published, e.g., CPA (Critical Path and Area-based scheduling) [4], or MCPA [5] (Modified CPA). Both algorithms divide the scheduling process into two independent steps, the allocation step and the mapping step. The MCPA algorithm takes special care when allocating processors to concurrent tasks to avoid a sequentialized execution of these tasks. Therefore, MCPA bounds the number of processors that can be allocated to a task, considering the total number of processors allocated all tasks of the same precedence layer.

Even though MCPA performs well for regularly shaped DAGs it may fail to achieve a small makespan as the number of concurrent tasks increases. In the worst case, there are more concurrent tasks than processors in the system, in which

[‡]This work was supported by a fellowship within the postdoctoral program of the German Academic Exchange Service (DAAD).

MCPA will not lead to an efficient schedule.

The contribution of this article is to show how the quality of two-step scheduling algorithms can be improved by using low-cost adjustments in both steps. It is demonstrated how the allocation step of MCPA can be adapted to deal with a more heterogeneous set (in terms of execution time) of concurrently executable tasks. We also analyze the impact of additional optimizations in the mapping step such as *packing of allocations* or *conservative backfilling*.

In the remainder of the paper, we first outline different scheduling algorithms that work in two steps (allocation and mapping) in Section II. Then we discuss the advantages and disadvantages of these approaches in Section III. In Section IV, we propose several low-cost improvements to MCPA and CPA. The experimental setup is described in Section V and the evaluation results are summarized in Section VI. We discuss related work in Section VII before the conclusions are drawn in Section VIII.

II. Two-level task scheduling in a nutshell

First, let us formally define the scheduling problem. As already introduced, a mixed-parallel application can be represented as a directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i \,|\, i = 1, \ldots, V\}$ is a set of nodes that represent tasks and $\mathcal{E} = \{e_{i,j} \,|\, (i,j) \in \{1, \ldots, V\} \times \{1, \ldots, V\}\}$ is a set of edges representing the communication dependencies between tasks. On the platform side, we consider a system that is composed of P identical processors interconnected by a network, so that each pair of processors can communicate. The tasks in DAG $\mathcal{G}$ are moldable, i.e., they can be executed by more than one processor. The time to complete a task $v \in V$ using p, processors is defined as $T(v,p)$, $1 \leq p \leq P$.

The question is then how to schedule all tasks of the DAG to the homogeneous set of processors such that the resulting makespan is minimized. For each task there are two important decisions to make: (1) how many processors should be assigned to this task, and (2) to which subset of the idle processors should the task be mapped.

Several algorithms have been developed to solve this problem. As for all other algorithms for NP-complete problems, there is always a trade-off between the accuracy of the solution and the time spent to compute it. In the case of scheduling mixed-parallel applications, the makespan can be reduced by increasing the time spent on optimizing the schedule. The so-called *one-step algorithms* allocate processors to tasks and perform the mapping of the allocations to the system in a single step, e.g., the algorithm iCASLB [6]. Hence, they integrate all information about the current status of the system and the DAG into the next scheduling decision. In order to reduce the complexity of M-task scheduling, *two-step algorithms* have been proposed. They achieve a good trade-off between the time spent to create the schedule and the quality of the resulting makespan. These algorithms decouple allocation or processors and mapping of tasks into two separate steps. Thus, in the first step, the allocation procedure determines on how many processors a task should be executed. In a second step, the algorithm computes a mapping of these allocations to the parallel platform. Compared to one-step algorithms, the main advantage of two-step algorithms is a lower computational complexity to find an appropriate solution, which makes them attractive to be used in a production environment.

Algorithm 1 *CPA allocation procedure

```
1: for all $v \in \mathcal{V}$ do
2:     $p(v) \leftarrow 1$
3: while $T_{CP} > T_A$ do
4:     $v \leftarrow$ select a task on the critical path that maximizes
5:         $\left(\frac{T(v,p(v))}{p(v)} - \frac{T(v,p(v)+1)}{p(v)+1}\right)$
6:         and $prec\_alloc(v) < P$ // for MCPA only
7:     $p(v) \leftarrow p(v) + 1$
8:     Update $T_A$ and $T_{CP}$
```

III. Properties of different two-level algorithms

This section recalls several two-level algorithms and discusses their applicability to specific use cases.

One of the best known two-level scheduling algorithms for mixed-parallel applications is CPA (Critical Path and Area-based scheduling) [4]. This algorithm mainly focuses on the allocation step, i.e., determining the size of an allocation for a given task. The basic idea of CPA is to find a good trade-off between the number of processors allocated to the tasks and the length of the critical path. The critical path T_{CP} is longest path from the source node to the target node of a DAG, i.e., the sum of the execution times of the nodes along this path. Another metric used in CPA is the term T_A, which is defined as $T_A = \frac{1}{P} \sum_v (T(v, p(v)) \cdot p(v))$. The time (or area) T_A is a measure of how much work a processor has to do on average. Both, T_{CP} and T_A are theoretical lower bounds of the makespan. The allocation procedure of CPA is shown in Algorithm 1. The allocation procedure starts with allocating one processor to each task (lines 1-2). Then it selects the task on the critical path that maximizes the equation in line 5, which is the task that benefits the most from an additional processor. The allocation of the task with the greatest benefit is increased by one and the values of T_{CP} and T_A are updated (lines 7-8). This iterative process continues until T_{CP} is smaller than T_A. The procedure starts with the greatest possible length of the critical path since all tasks are assigned to a single processor. By allocating more processors to a task on the critical path, the length of the critical path decreases but the size of the area covered by the tasks also increases.

This allocation procedure requires a performance model of the tasks' execution time that is monotonically decreasing with the number of processors, i.e., a task is executed faster if it is assigned to more processors. The execution time of multiprocessor tasks is modeled using Amdahl's law, which says that the execution time of a parallel program is composed of a sequential part and a parallel part, and only the time spent in the parallel part can be reduced. Thus, the execution time of a task v on p processors depends on the sequential part α of v,

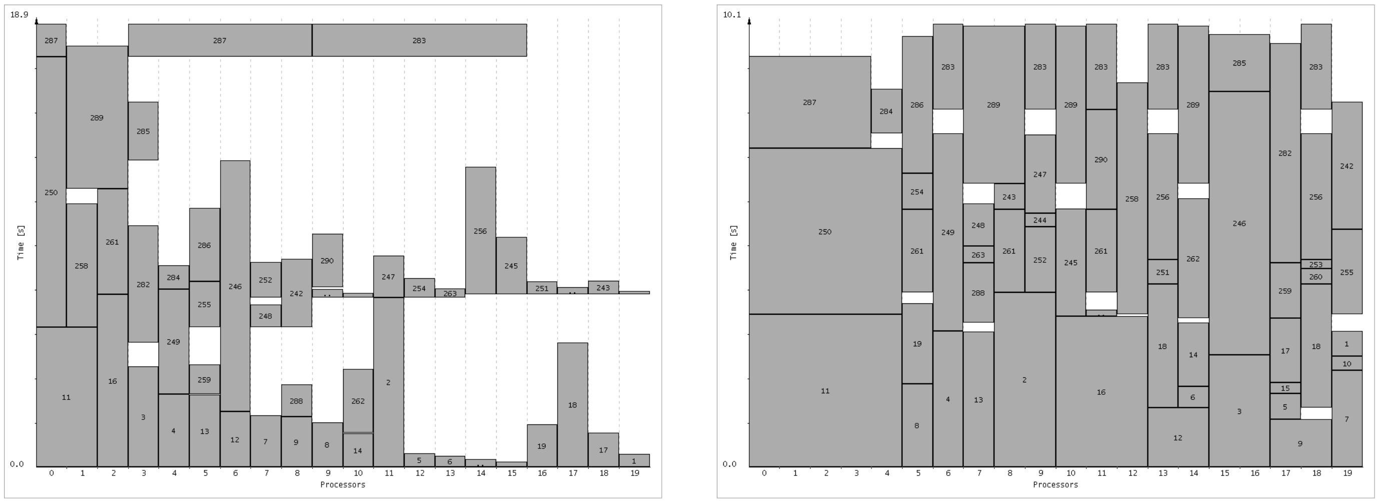


Fig. 1. Comparison of schedules produced by MCPA (left) and CPA (right) for a DAG with 50 computation tasks executed on a cluster comprised of 20 processors.

$0 \leq \alpha \leq 1$, leading to $T(v,p) = \left(\alpha + \frac{1-\alpha}{p}\right) \cdot \tau$. The variable τ represents the sequential execution time of task v. So, the performance model based on Amdahl's law is monotonically decreasing with the number of processors.

In the second step, the mapping step, CPA sorts the ready tasks by decreasing priority (e.g., bottom level of a task) and schedules them using a list scheduling approach. For a selected task v, the first $p(v)$ processors that become idle will be assigned to this task.

Drawbacks of CPA have been highlighted in [5] and [7]. For some application and platform configurations, CPA may allow allocations to grow too big, which limits the task-parallelism and therefore the overall performance. The MCPA algorithm in [5] avoids these large allocations by preventing allocations of tasks of one precedence level to exceed the total number of processors. This improvement works well for layered or regular graphs where concurrently executable tasks have similar costs (number of operations to perform). Thus, MCPA uses the same allocation procedure as shown in Algorithm 1. Only the test in line 6 ensures that the number of processors of a layer does not exceed the number of processors of the system. So, MCPA tends to favor task-parallelism over data-parallelism.

Another algorithm that uses a similar allocation procedure is HCPA [8]. In contrast to CPA and MCPA, HCPA targets a set of homogeneous clusters (multi-cluster) as execution platform. HCPA uses another definition of the average area T_A in order to stop the allocation procedure before producing large allocations. For HCPA, the average area is defined as $T_A = \frac{1}{\min(P, \sqrt{V \times P})} \sum_i W(v_i)$.

An issue common to the three algorithms described above has been raised and addressed in [9]. The totally decoupled allocation and mapping procedures of two-step scheduling algorithms may cause unnecessary data redistributions. For instance, subsequent tasks may have close but different allocations (e.g., 15 and 16) that may imply a complex data redistribution that could be avoided. Moreover, because of possible contention on network links, data redistributions may delay the start time of tasks, causing the actual schedule to differ significantly from the predicted schedule. During the mapping step, the RATS algorithm can reconsider the allocations determined to minimize the impact of the data redistributions.

IV. Low-cost improvements of MCPA

The previous section introduced different algorithms that use two steps to schedule mixed-parallel applications. As we have seen, each of the algorithms CPA, MCPA, and HCPA has strengths and weaknesses for certain DAG or platform configurations. Therefore, we set out to categorize the advantages and disadvantages with the goal to find a more adaptive algorithm that would produce efficient schedules in most cases. Thus, we ran a huge number of experiments with different types of DAGs on varying platforms. The experiments were conducted using SimGrid, which will be discussed in Section V.

Surprisingly, the results revealed that MCPA leads to good schedules for most of the DAGs and platforms. However, several cases could be found in which MCPA produced a significantly worse schedule compared to the schedules of the competitors. Is has been observed that MCPA performs inferior to CPA if the DAGs become more irregular. Fig. 1 pictures the problem of MCPA for more irregularly structured DAGs. Recall that MCPA avoids allocating more than P processors to a layer of tasks, where P is the total number of processors in the system. On a fairly small cluster with only 20 nodes as seen in this figure, MCPA does not allocate more than one processor to a task if the current layer contains 20 or more tasks. This would not be a big problem if all tasks of a layer have a similar execution time. But as irregularity increases this assumption does not hold anymore. The right-hand side of Fig. 1 shows the schedule produced by CPA. Since it does not consider the total number of processors already assigned to a layer, it creates a far better schedule. However, the strategy of MCPA to limit the number of processors of a precedence layer is efficient for regular DAGs, such as FFT DAGs or Matrix-Multiplication DAGs [5]. Therefore, we need a hybrid version of both allocation procedures that loosens the restrictions of MCPA for more irregular DAGs in order to achieve a more balanced workload.

By evaluating the quality of the mapping phase, we also found numerous cases where the start time of tasks had been delayed because not enough idle processors were available. But the number of available processors was only slightly different from the allocation size of the tasks, so that packing of allocations in the mapping phase could also be an option to improve the resulting schedule. The packing of allocations should reduce the fragmentation of the schedule, and therefore lead to a smaller makespan.

A. Improving the allocation procedure

As stated before, the allocation procedure of MCPA only allocates as many processors to tasks of one precedence layer as there are processors in the system. If there are many concurrent tasks ready to be executed and they have different computational costs, the allocation procedure should not restrict the growth of allocations of nodes on the critical path. That means that if there are several tasks in one layer and the tasks significantly differ in criticality (bottom level) then we should allow more processors to be assigned to the critical tasks. In other words, if we find a very critical task in a layer and the number of processors in this layer is bounded by P we should expand the layer.

That leads to the question when we should expand the processor limit of one precedence layer. An easy answer would be: whenever there is some work imbalance in the layer. An imbalanced layer is one where a following layer is delayed by one huge task and this space cannot be filled with other tasks, as seen in Fig. 1. To solve this problem, we define a variable cr that denotes the cover ratio of a layer. The cover ratio is defined as the fraction of the sum of the work done by all tasks of a layer and the minimum height of a layer. The work done in a layer L is $W_L = \sum_{w \in L} W(w)$. The minimum area L_A that is required to execute the layer L is given by the maximum height (longest execution time) of a task in L and the total number of processor P, $L_A = h_v^{min} \cdot P$. With these variables defined, the cover ratio $cr, 0 \leq cr$ is given as $cr = \frac{W_L}{L_A}$. For a highly imbalanced layer the cover ratio will be much smaller than 1.

But this ratio will not be enough to detect if a layer is imbalanced because MCPA restricts the allocation of processors or because the layer contains only a few tasks. For that reason, we introduce another heuristic parameter wr, $0 < wr \leq 1$ that defines the minimum number of tasks that have to be in a layer in order to loosen the restrictions when allocating processors. The parameter is defined as $wr = \frac{V_{min}}{P}$, where V_{min} denotes the minimum number of nodes that a layer has to contain in order to drop the restrictions. So, wr is only a scaling factor of P. In the actual algorithm, the value wr is predefined by the user and the value of V_{min} can be computed.

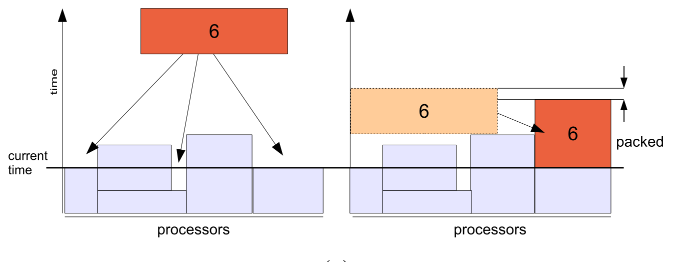
Algorithm 2 MCPA2 allocation procedure

```
1: for all $v \in \mathcal{V}$ do
2:     $p(v) \leftarrow 1$
3:     $pl(v) \leftarrow DFS\_DEPTH(v)$
4: for all $v \in \mathcal{V}$ do
5:     $prec\_alloc(pl(v)) \leftarrow prec\_alloc(pl(v)) + p(v)$
6: for all $l \in PL$ do
7:     $prec\_p(l) \leftarrow p$
8: while $T_{CP} > T_A$ do
9:     $V_c \leftarrow \{$set of currently critical tasks$\}$
10:    for all $v \in V_c$ do
11:        $bf(v) \leftarrow \left(\frac{T(v,p(v))}{p(v)} - \frac{T(v,p(v)+1)}{p(v)+1}\right)$
12:    sort $V_c$ by decreasing $bf(v)$
13:    $v_b \leftarrow None$
14:    for all $v \in V_c$ do
15:        if $prec\_alloc(v) < prec\_p(pl(v))$ then
16:            $v_b \leftarrow v$
17:        else
18:            if $|lp(v)| \geq wr \cdot P$ then
19:                $W_L \leftarrow \sum_{w \in lp(v)} W(w)$
20:                $h_v^{min} \leftarrow \max_{w \in lp(v)}(T(w, p(w)))$
21:                $cr \leftarrow W_L/(h_v^{min} \cdot p)$
22:                if $cr < cr_{min}$ then
23:                    $prec\_p(pl(v)) \leftarrow 2 \cdot prec\_p(pl(v))$
24:                    $v_b \leftarrow v$
25:        if $v_b \neq None$ then
26:            break
27:    if $v_b \neq None$ then
28:        $p(v) \leftarrow p(v) + 1$
29:        Update $T_A$ and $T_{CP}$
30:    else
31:        break
```

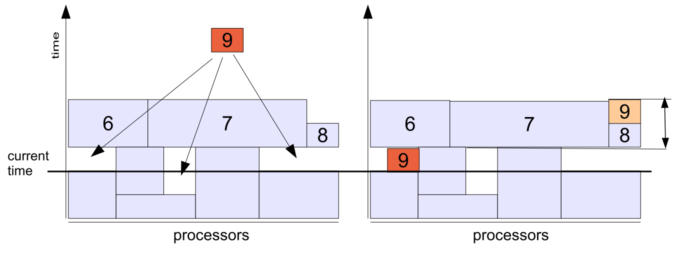
TABLE I
NOTATION FOR THE SCHEDULING ALGORITHMS.

PL	set of precedence levels
$p(v)$	allocation of node v
$T(v, p(v))$	execution time of v with $p(v)$ processors
T_A	average time that a processor is busy
T_{CP}	length of the critical path
$W(v)$	work (area) when executing v
$pl(v)$	precedence level of node v
$prec\_alloc(l)$	for precedence layer l
$bf(v)$	benefit of node v when allocated an additional processor
$lp(v)$	nodes in the same precedence layer as v
h_v^{min}	minium height of the precedence layer of v

After having the heuristic parameters cr and wr defined, we can now introduce the adapted pseudo-code of MCPA. The new allocation phase is referred to as MCPA2 and presented in Algorithm 2. Variables and functions used in the pseudo-code are defined in Table I. The algorithm starts with the allocation of one processor to each task. After that the total number of processors per layer is accumulated (lines 4-5). The algorithm also uses the function $prec\_p(l)$ that returns the current processor bound for a layer l. In the beginning, all precedence layers are bounded by P processors (lines 6-7). Then, as done by MCPA, the new procedure iteratively assigns processors to tasks by selecting the task with the greatest



(a)



(b)

Fig. 2. Mapping strategies: (a) packing allocations, and (b) backfilling of tasks.

benefit (while loop, line 8). In each iteration, the current benefit is computed for each task on the critical path, i.e., by how much would the execution time of a task be reduced if an additional processor is assigned. The critical tasks are then sorted by decreasing benefit. Hence, the task with the greatest benefit will be inspected first. After a task v is selected (line 14), MCPA2 verifies that the number of processors in the same layer does not exceed the number of processors in the system. If the layer is not completely filled, we can add one more processor to the task with the greatest benefit. If not, the algorithm checks that there are enough tasks in the same layer to justify adding more processors to the current layer (line 18). If so, the current cover ratio is computed and checked against a minimum cover ration that the user defined (cr_{min}). In case this layer is less covered with work than specified by cr_{min} we double the number of processors bound to this layer, i.e., adding another virtual layer within this layer. This algorithms repeats until $T_{CP} \leq T_A$ or none of the tasks matches any of these conditions.

B. Improving the mapping function

The mapping step is another subject to be investigated. A good example is the RATS algorithm (redistribution-aware two-step scheduling) [9]. While mapping tasks, the RATS algorithm dynamically changes the size of allocations in order to reduce the redistribution costs between subsequent tasks. It has been shown that re-adjusting the allocations in the mapping step can lead to a smaller makespan.

In the context of this paper, we want to assess how much we can gain from an additional adjustment phase in the mapping step. We want to investigate two possible optimizations that could be applied to the mapping procedure of CPA and MCPA: *packing* of allocations and *backfilling* of tasks. Fig. 2 illustrates these two possible optimizations. On the left-hand side of this figure, the strategy of packing allocations is depicted. At a certain time (current time in figure), the task 6 becomes ready and should be mapped to the cluster. If the current size of the allocation is kept constant, task 6 can only start at the current time since there are not enough processors available. However, if the size of the allocation were packed, the task could start immediately. But this will increase the execution time of this task, so that the expected finish time has to be compared to the original expected finish time. If the task finishes earlier using the packed allocation, packing will in general lead to less fragmentation in the schedule. In order to keep the computational complexity of packing low, we simply check if the task's projected finish time with all currently available processors is smaller than the projected finish time with the original allocation. Another (more complex option) would be to reduce the allocation size to a minimum in order to increase the chance to squeeze in another task.

Another possible method for optimization is the use of conservative backfilling, i.e., starting tasks with lower priority before tasks with high priority, if high priority tasks cannot be executed due to an insufficient number of resources. This technique is illustrated in Fig. 2. At the current time, tasks 6, 7, 8, and 9 become ready, but 6, 7, and 8 have a higher priority than task 9. Conservative backfilling attempts to find a hole in the schedule between the current time and the starting time of the higher priority tasks. If a hole is found and task 9 fits into the hole without delaying the start of the other, then it will be executed. The problem of conservative backfilling is that it adds computational complexity to the mapping phase. The algorithm has to keep a list of idle times of all processors. To find a hole, the idle times of all processors have to be checked for a given time slot. Since a hole can be really huge, we cannot simply assigned all idle processors of that hole to the current task. Therefore, the number of processors assigned to this task is optimized by using a bisection method, so that we can find the smallest possible allocation inside the hole in at most $O(\log P)$ steps.

V. EXPERIMENTAL SETUP

We use a simulator to compare and evaluate the algorithms. Simulation allows us to perform a statistically significant number of experiments for a wide range of application and platform configurations. We use the SIMGRID toolkit [10] as the basis for the simulator. SIMGRID provides the required fundamental abstractions for the discrete event simulation of parallel applications in distributed environments and was specifically designed for the evaluation of scheduling algorithms.

A. Platforms

We consider two clusters of the Grid'5000 platform[1]. The cluster *Chti* is located in Lille, while the other cluster *Grelon* is located in Nancy. The cluster *Chti* comprises

[1]http://www.grid5000.fr

20 nodes (Opteron 252, 2.6 GHz) with a computing power of 4.3 GFlop/s. The cluster *Grelon* consists of 120 nodes (Xeon, 1.6 GHz), each with a computing power of 3.2 GFlop/s. The computing power was obtained with the HP Linpack benchmark using the ACML routines [11]. Each cluster uses internally a Gigabit Ethernet for interconnections (100 μs latency and 1 Gb/s bandwidth). The *Grelon* cluster is divided into five cabinets, each comprising 24 nodes, making its network hierarchical. We have been using these clusters for the following reasons: (1) We have used them as testbeds in previous articles on mixed-parallel scheduling. So, we have good insight why effects may show up. (2) They are used in production.

It is the main purpose of cluster *Chti* to trigger the allocation problem of MCPA when many concurrent tasks are executable, while *Grelon* was primarily chosen to show that our proposed heuristics will not cause poor performance on larger clusters.

B. Applications

To instantiate the model of mixed-parallel applications described in Section II, we need to pinpoint the functions for predicting the execution time of data-parallel tasks and also the type of task graphs that should be evaluated.

To model the execution time of data-parallel tasks we assume that a task operates on a dataset of d double precision (8 bytes) elements (for instance a $\sqrt{d} \times \sqrt{d}$ square matrix). We assume that processors have 1 Gb of memory and thus $d \leq 125 \times 10^6$. We also assume that d is above 4×10^6 (if d is too small, the data-parallel task should be merged with its predecessor or successor). We model the computational complexity of a task (number of operations) with the following expression, which is representative of common applications: $a \cdot d$ (e.g., a stencil computation on a $\sqrt{d} \times \sqrt{d}$ domain, or an image filter). The complexity parameter a is picked randomly between 2^6 and 2^9, to capture the fact that some of these tasks often perform multiple iterations.

The latter gives us only the number of operations, but not the scalability of each task. Since we use Amdahl's law to model the execution time of parallel tasks, we have to determine the parameter α, which defines the non-parallelizable (sequential) fraction of the program. We pick a random value for α uniformly between 0 % and 25 %. Thus, the execution time of tasks strictly decreases as the number of processors increases.

We consider applications that consist of 25, 50, or 100 moldable tasks. We use four popular parameters to define the shape of the DAG: width, regularity, density, and "jumps". 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 leads to "fork-join" graphs. The regularity denotes the uniformity of the number of tasks in each level. A small value means that levels contain very dissimilar numbers of tasks, while a large value means that all levels contain similar numbers of tasks. The density denotes the number of edges between two levels of the DAG, with a small value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. In our experiments we use values 0.2 and 0.8. We generate a first set of *layered* DAGs using these three parameters with the particularity that all the tasks in a given level have the same cost. By generating three samples for each DAG configuration and accounting to the four aforementioned computational complexity scenarios, we obtain a total of 108 layered random DAGs.

TABLE II
RANDOM DAG GENERATION PARAMETERS AND VALUES.

	Layered	**Irregular**
#computation tasks	{ 25, 50, 100 }	
non-parallelizable fraction (α)	[0.0; 0.25]	
width	{ 0.2, 0.5, 0.8 }	
density	{0.2, 0.8 }	
regularity	{ 0.2, 0.8 }	
jump length	$\emptyset$	{ 1, 2, 4 }
#samples	3	
Total	108	324

We also generate another set of irregular DAGs in which tasks in the same level can have different costs. Furthermore, we add random "jump edges" that connect level l with level $l+jump$, for $jump = 1, 2, 4$ (the case $jump = 1$ corresponds to no jumping "over" any level). The details can be found in the documentation of the DAG generation program [12]. This generator is available to ease the reproduction of the presented results. We have 108 different irregular DAG types and 3 samples per type, for a total of 324 irregular DAGs. A summary of the generated layered and irregular random DAGs can be found in Table II.

While the above specifies a way to generate a set of synthetic DAGs, we also want to consider DAGs for the Strassen matrix multiplication and for the Fast Fourier Transform (FFT) application. Both are classical test cases for DAG scheduling algorithms and we refer the reader to [13] for more details. These DAGs are more regular than our synthetic DAGs, which are more representative of workflow applications that compose operators in arbitrary ways. We consider FFT DAGs with 2, 4, and 8 levels (that is 5, 15, and 39 tasks), while all the Strassen DAGs have 25 tasks. As for the random DAGs, we consider 4 different computational complexity scenarios. The purpose of this is to explore scenarios beyond those corresponding to the actual FFT and Strassen applications. We generate 25 samples for each parameter combination leading to 100 FFT DAGs and 25 Strassen DAGs.

It was mentioned before that the edges between nodes represent data dependencies. However, in this work, communication costs between tasks are not considered, neither during the scheduling nor during the simulation. So, we simply use precedence constraints to model the DAGs. The main reason for not considering the communication time between tasks is that comparing the quality of the scheduling algorithms would be harder, especially the allocation phase. Communication costs would add another level of complexity to the model, which would lead to more "noise" in the results.

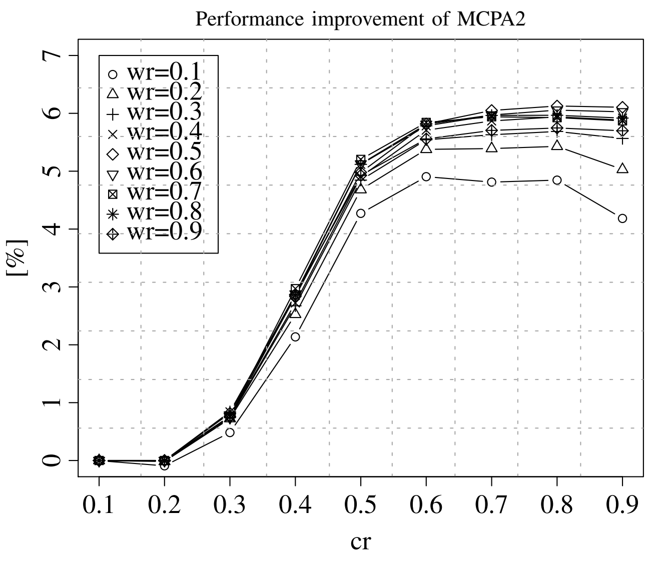


Fig. 3. Tuning wr and cr empirically. Average relative makespan of MCPA2 (to MCPA) for irregular DAGs on cluster *Chti*.

VI. Experimental results

The effectiveness of the proposed optimizations is evaluated in different steps. At first, we need to detect a pair of heuristic parameters (cr, wr) for MCPA2 that shows good performance in the allocation phase. After that, we evaluate the different optimizations of the mapping step by comparing them to the standard mapping procedure. Finally, we compare the overall performance of the best heuristics for each step to the scheduling performance of the original MCPA algorithm.

A. Tuning the allocation step

In the first experiment we want to determine good values for the heuristic parameters cr (cover ratio) and wr (DAG width ratio). In this experiment all types of DAGs (Strassen, FFT, layered, random) were scheduled using MCPA and MCPA2 onto cluster *Chti*. For cr and wr we used all combinations of discrete values from the set $\{0.1, 0.2, \ldots, 0.9\}$. It was not the main objective to find the optimal parameter pair. In fact, we wanted to identify the general trends. In total, about 46,000 experiments were conducted.

Fig. 3 shows the results for irregular DAGs. In this graphic, the average relative makespan of MCPA2 compared to MCPA is plotted for all pairs of cr and wr. After also evaluating the results for the other DAG families, we picked the values $cr = 0.8$ and $wr = 0.6$ for further experiments. These values have led to a small average makespan for all types of DAGs.

One could also determine the optimal parameter pair for each type of DAG by inspecting the structure of the DAG. The advantage would be that MCPA2 would be mostly on par with or better than MCPA for all the corner cases of MCPA. However, we primarily want to show that modifying the allocation step and using reasonable parameters leads to a better overall performance.

B. Tuned allocation procedure

Having the values for cr and wr set, we can compare the allocation procedures of CPA, MCPA, and MCPA2. In this experiment all two-step algorithms use the same standard mapping strategy, which takes the first $p(v)$ processors that become available. Thus, the scheduling performance is driven by the quality of the allocation procedure.

Fig. 4 shows the experimental results for the clusters *Chti* and *Grelon*. The charts show for each allocation procedure the average degradation from the best algorithm on the y-axis. Simply comparing the average makespan does not work since the lengths of the DAGs differ. Therefore, the makespan is normalized for each experiment to the smallest makespan produced by one of the algorithms. The degradation from this best algorithm is accumulated for each algorithm and later divided by the number of experiments. Thus, the smaller the bar the better the algorithm on average. It can be seen that CPA performs worst in most of the cases on both clusters. However, it performs better than MCPA on *Chti* for irregular DAGs. But that was the corner case that we set out to fix in MCPA. Comparing MCPA2 to the others, we see that MCPA2 performs well in all cases. Only in the case of layered DAGs on *Chti* (top) MCPA shows a slightly better performance. But this difference is only about 1% on average, which is almost insignificant. The final result of this comparison is that MCPA2 performs better than MCPA for irregular DAGs on smaller clusters like *Chti*, and also performs well in the other cases.

C. Mapping strategies

We have implemented two different optimizations for the mapping procedure of CPA and MCPA, packing of allocations and conservative backfilling. We could exclusively enable them to see possible performance differences. However, we tested only three cases: (1) the standard mapping procedure; (2) the mapping procedure that also checks if allocations can be packed (suffix “p”); and (3) a mapping procedure that first checks if an allocation can be backfilled and, if not, tries to pack it (suffix “pb”). The reason is that packing has the biggest impact on the schedule, and if it works well then backfilling will have almost no effect as only small holes may appear.

Fig. 5 compares the average degradation from the best algorithm for all DAG families on cluster *Chti*. There are three bars for each algorithm for a total of nine bars. The first three represent the results of the allocation algorithm CPA with standard mapping, packing enabled, and packing + backfilling enabled. The other six bars present the results for MCPA and MCPA2. It can be observed that packing always reduces the degradation from the best for all allocation procedures. So, we can reason that the packing of allocations leads to a smaller makespan by requiring only a low complexity computation. With our set of DAGs and platforms we could not observe significant improvements of the makespan when conservative backfilling was enabled. Backfilling can also lead to unwanted side effects. In some cases, a high priority task may depend on a low priority task that can be executed prior to its original starting time. Thus, scheduling this low priority task may make

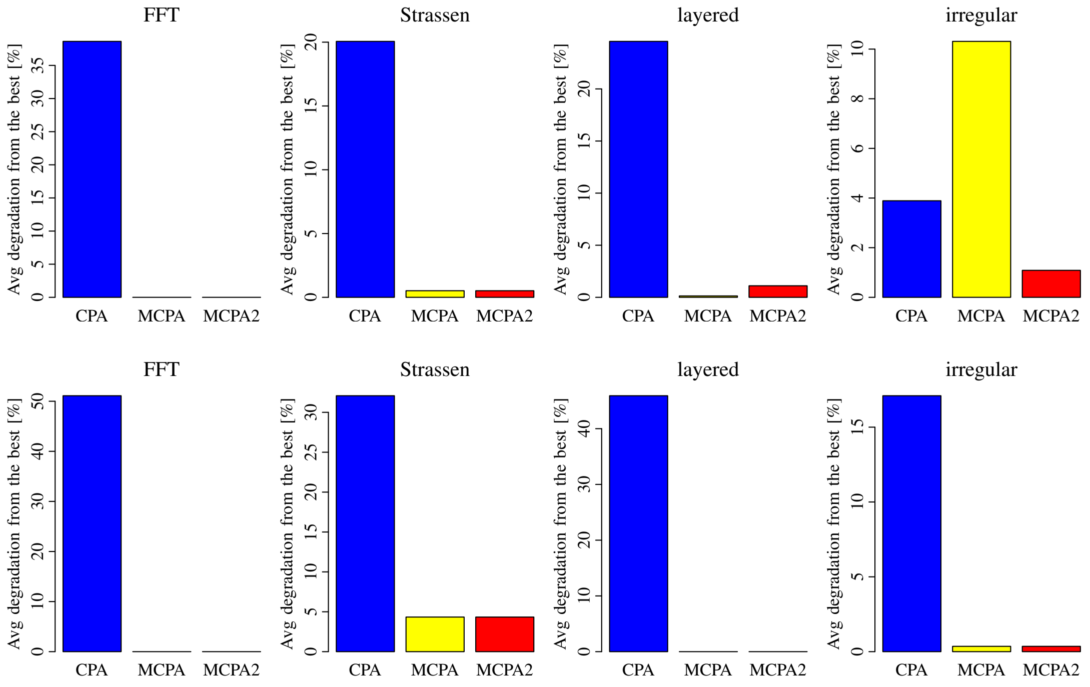


Fig. 4. Comparison of the average degradation from the best makespan for CPA, MCPA, and MCPA2 on the systems *Chti* (top) and *Grelon* (bottom).

other high priority tasks become ready earlier, which could change the original processing order of tasks. This side effect has sometimes led to a bigger makespan, sometimes to a lower. Since applying a conservative backfilling strategy has not led to significant performance improvements and it is costlier than simple packing, we believe that packing will lead to the best trade-off between the time to compute a mapping and the resulting makespan.

In the final experiment we want to compare the performance of the composite algorithm of the tuned allocation procedure and the tuned mapping procedure to the standard version of MCPA. Thus, in this experiment MCPA only uses the standard mapping procedure. This allows us to assess the overall improvement that we gain by optimizing both algorithmic steps. Fig. 6 presents the results of this experiment for the clusters *Chti* (left) and *Grelon*. We only show the graphs for the Strassen DAGs and the irregular DAGs since the bars for the other DAG families (FFT, layered) were almost zero for both algorithms. In addition, a complete summary of the scheduling performance of MCPA2 for all DAG families is given in Table III. As can be observed in Fig. 6, MCPA2 reduces the average makespan for Strassen and irregular DAGs significantly. As seen in the section on tuning the allocation procedure (Section VI-B), the improvement in makespan for irregular graphs on *Chti* is mainly due to the better allocation procedure of MCPA2. However, this allocation procedure alone did not have an impact on the scheduling performance on *Grelon* before. But by having packing enabled, the average scheduling performance of MCPA2 is superior to MCPA for these irregular DAGs. The packing of allocations also leads to a smaller average makespan for the Strassen DAGs on both clusters. Table III also shows the minima, maxima, and the standard deviation of the degradation from the best metric, recorded for MCPA2 for each DAG family. We can see that schedules produced by MCPA2 can be up to 65% shorter than the ones of MCPA for irregular DAGs. It can also be noted that the scheduling performance of FFT and layered DAGs is almost equal for both algorithms. This is due to the regular structure of both DAG families, where nodes on one layer have very similar costs (number of operations to perform). In these cases, MCPA already performed well. The more irregular the DAGs are the better MCPA2 becomes in terms of makespan. Thus, the experimental data have shown that the tuned allocation procedure of MCPA2 and the improved mapping step (with packing enabled) increase the performance of the scheduler on average. The new algorithm gains about 10-15% on average for Strassen DAGs and irregular DAGs.

VII. Related work

Many algorithms for scheduling mixed-parallel programs onto parallel systems consider static DAGs. Several scheduling algorithms have been designed for the case of homogeneous parallel platforms, e.g., CPR [14] and CPA [4]. The iterative algorithm CPR attempts to allocate one processor to an M-task on the critical path. As a consequence, as more idle processors are allocated to an M-task on the critical path per iteration, the execution time of the whole program is reduced.

The one-step algorithm iCASLB was shown to lead to better performance than the two-step algorithms but at the price of a higher complexity [6]. This algorithm performs the allocation and mapping of tasks in one step by iteratively increasing the allocations of tasks on the critical path. It also uses a look-

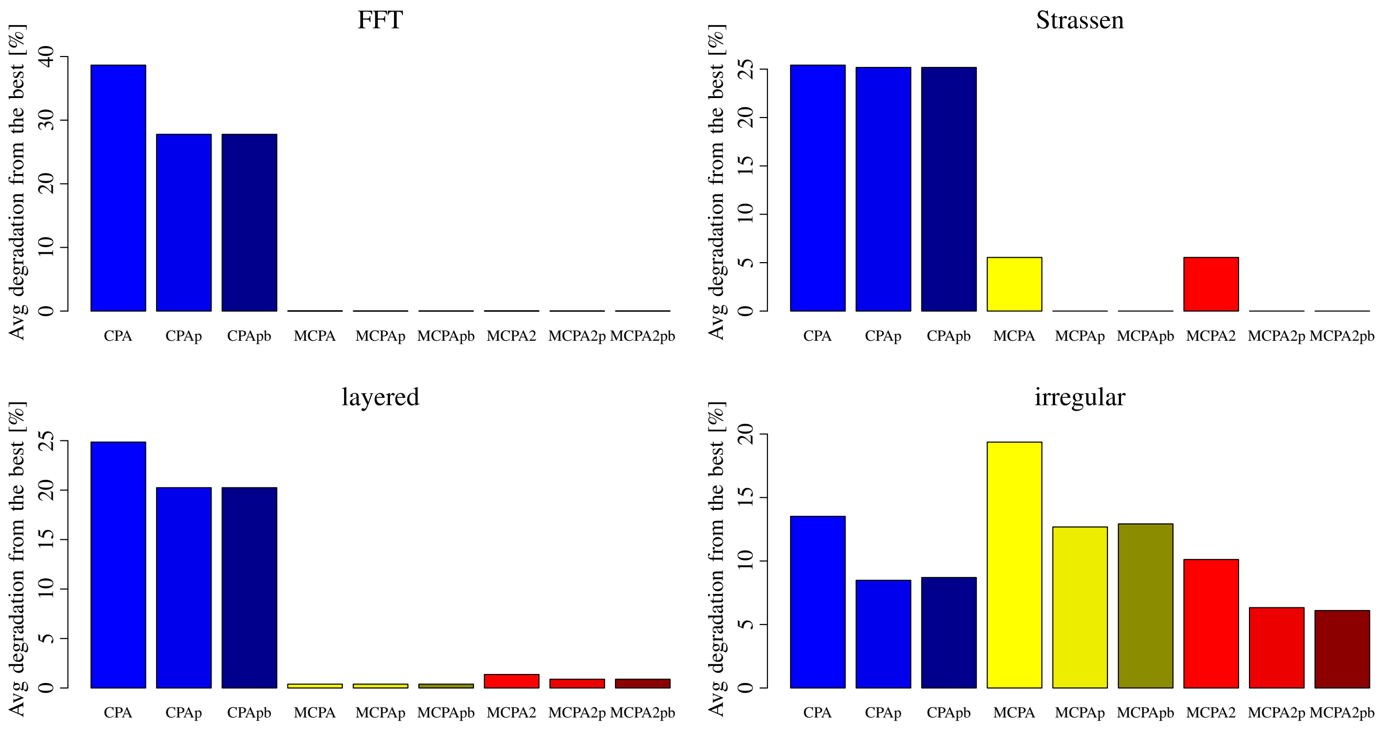


Fig. 5. Comparison of the average degradation from the best makespan for CPA, MCPA, and MCPA2 on cluster *Chti* for different DAG types. The suffix "p" denotes that packing has been enabled, and "pb" denotes that packing and backfilling has been enabled.

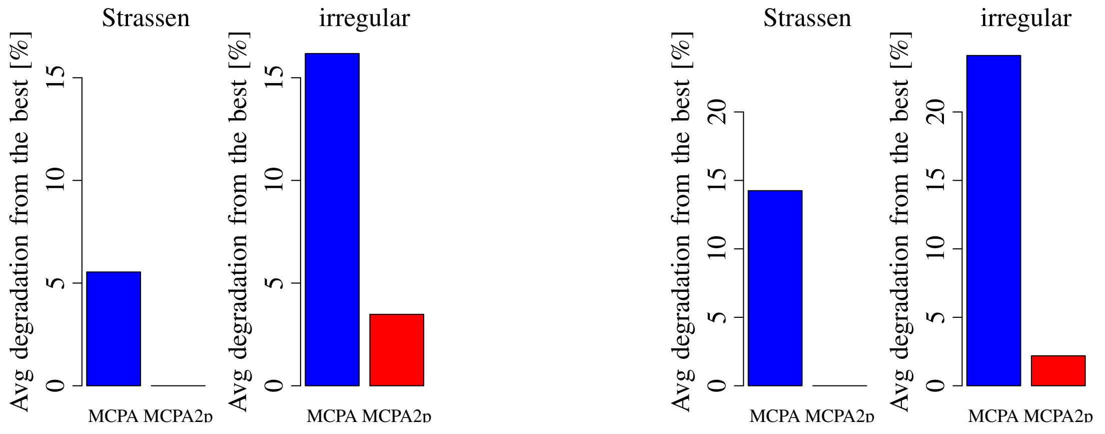


Fig. 6. Comparison of the *overall scheduling performance* of MCPA and MCPA2 (packing enabled) for different types of DAGs on *Chti* (left) and *Grelon*.

ahead mechanism to avoid becoming trapped in local minima and a backfilling approach to improve the schedule.

The problem of scheduling mixed-parallel workflows using advance reservations on a parallel platform has been studied in [19]. The authors compare several algorithms that reuse ideas from the CPA algorithm to bound task allocations and to compute the most resource-conservative allocation.

Mixed-parallel task scheduling for more heterogenous environments have recently been developed, e.g., HCPA [7] and MHEFT [15]. A good overview of these algorithms as well as a discussion of modifications of MHEFT to gain better performance are summarized in [7]. The Δ-CTS [16] attempts to increase the effective degree of task-parallelism of a DAG by relaxing the criticality classification of ready tasks (bottom levels).

Lately many researchers have focused on scheduling scientific workflows onto multi-clusters or computational grids [17]. In most of these case, the structure of the DAGs are dynamically defined during the execution of the workflows. Hence, algorithms are required to dynamically schedule ready nodes onto the system [18].

TABLE III
SCHEDULING PERFORMANCE OF MCPA2 COMPARED TO MCPA (MIN, MAX, MEAN, AND SD ARE BASED ON THE DEGRADATION FROM BEST METRIC).

cluster	family	#dags	wins	draws	min [%]	max [%]	mean [%]	sd [%]
chti	FFT	100	5	94	0.00	0.06	0.00	0.01
	Strassen	25	14	11	0.00	0.00	0.00	0.00
	layered	108	10	88	0.00	21.83	0.84	3.24
	irregular	324	202	30	0.00	65.86	3.48	8.15
grelon	FFT	100	7	93	0.00	0.00	0.00	0.00
	Strassen	25	11	14	0.00	0.00	0.00	0.00
	layered	108	0	108	0.00	0.00	0.00	0.00
	irregular	324	202	74	0.00	52.36	2.19	7.29

VIII. CONCLUSIONS

This article has focused on the improvement of low-complexity algorithms for scheduling mixed-parallel applications onto a homogeneous clusters. The scheduling algorithms that were investigated work in two steps. In the first step, the algorithm defines how many processors are assigned to a parallel task. In the second step, the algorithm maps these allocations to processors of the cluster. Several low-cost optimizations of the allocation step as well as of the mapping step have been proposed in this article. A new allocation procedure MCPA2 has been described and it has been shown that it leads to a smaller application makespan compared to MCPA when many irregular tasks are concurrently executable. It has been shown that packing allocations in the mapping step can also decrease the average makespan for all considered two-step algorithms. As an additional result, we have observed that conservative backfilling does not have a significant impact on the makespan in the considered cases.

REFERENCES

[1] S. Chakrabarti, K. Yelick, and J. Demmel, "Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs," *Models and Scheduling Algorithms for Mixed Data and Task Parallel Programs*, vol. 47, no. 2, pp. 168–184, 1997.

[2] S. Ramaswamy, E. W. H. Iv, and P. Banerjee, "Compiling MATLAB Programs to ScaLAPACK: Exploiting Task and Data Parallelism," *International Parallel Processing Symposium*, vol. 0, p. 613, 1996.

[3] H. Casanova and J. Dongarra, "NetSolve: A Network-Enabled Server for Solving Computational Science Problems," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, pp. 212–223, Fall 1997.

[4] A. Rădulescu and A. J. C. van Gemund, "A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling," in *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 69–76.

[5] S. Bansal, P. Kumar, and K. Singh, "An Improved Two-Step Algorithm for Task and Data Parallel Scheduling in Distributed Memory Machines," *Parallel Computing*, vol. 32, no. 10, pp. 759–774, 2006.

[6] N. Vydyanathan, S. Krishnamoorthy, G. Sabin, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz, "An Integrated Approach for Processor Allocation and Scheduling of Mixed-Parallel Applications," in *35th International Conference on Parallel Processing (ICPP'06)*, Colombus, OH, Aug. 2006, pp. 443–450.

[7] T. N'Takpé, F. Suter, and H. Casanova, "A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms," in *Proc. of the 6th Int. Symposium on Parallel and Distributed Computing*, 2007.

[8] T. N'Takpé and F. Suter, "Critical Path and Area Based Scheduling of Parallel Task Graphs on Heterogeneous Platforms," in *Proc. of the 12th Int. Conference on Parallel and Dist. Systems (ICPADS 2006)*, 2006, pp. 3–10.

[9] S. Hunold, T. Rauber, and F. Suter, "Redistribution Aware Two-Step Scheduling for Mixed-Parallel Applications," in *Proc. of the 10th IEEE Int. Conference on Cluster Computing (Cluster 2008)*, 2008.

[10] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experimentations," in *Proc. of 10th Int. Conf. on Computer Modeling and Simulation (UKSim)*, 2008.

[11] "AMD Core Math Library (ACML)." [Online]. Available: http://developer.amd.com/cpu/Libraries/acml/

[12] DAG Generation Program, http://www.loria.fr/~suter/dags.html.

[13] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[14] A. Rădulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker, "CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems." in *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*. IEEE Computer Society, 2001, p. 39.

[15] H. Casanova, F. Desprez, and F. Suter, "From Heterogeneous Task Scheduling to Heterogeneous Mixed Parallel Scheduling," in *Proceedings of the 10th International Euro-Par Conference (Euro-Par'04)*, ser. LNCS, M. Danelutto, D. Laforenza, and M. Vanneschi, Eds., vol. 3149. Pisa, Italy: Springer, August/September 2004, pp. 230–237.

[16] F. Suter, "Scheduling Δ-Critical Tasks in Mixed-Parallel Applications on a National Grid," in *8th IEEE/ACM International Conference on Grid Computing (GRID 2007), Proceedings*. IEEE, 2007, pp. 2–9.

[17] L. Meyer, D. Scheftner, J.-S. Vckler, M. Mattoso, M. Wilde, and I. T. Foster, "An Opportunistic Algorithm for Scheduling Workflows on Grids," in *Proceedings of the VECPAR'2006*, ser. Lecture Notes in Computer Science, vol. 4395. Springer, 2006, pp. 1–12.

[18] R. Prodan and T. Fahringer, "Dynamic Scheduling of Scientific Workflow Applications on the Grid: A Case Study," in *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA: ACM Press, 2005, pp. 687–694.

[19] K. Aida and H. Casanova, "Scheduling mixed-parallel applications with advance reservations," *Cluster Computing*, vol. 12, no. 2, pp. 205–220, 2009.