Scheduling Mixed-Parallel Applications with Advance Reservations

Kento Aida*
Center for Grid Research and Development
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo 101-8430, Japan
aida@nii.ac.jp

Henri Casanova†
Dept. of Information and Computer Sciences
University of Hawai'i at Manoa
1680 East-West Rd, POST 317
Honolulu, HI 96822, U.S.A.
henric@hawaii.edu

ABSTRACT

This paper investigates the scheduling of mixed-parallel applications, which exhibit both task and data parallelism, in advance reservations settings. Both the problem of minimizing application turn-around time and that of meeting a deadline are studied. For each several scheduling algorithms are proposed, some of which borrow ideas from previously published work in non-reservation settings. Algorithms are compared in simulation over a wide range of application and reservation scenarios. The main finding is that schedules computed using the previously published CPA algorithm can be adapted to advance reservation settings, notably resulting in low resource consumption and thus high efficiency.

1. INTRODUCTION

An effective way to extract the most performance and scalability out of a parallel application is to exploit both task parallelism and data parallelism. With this approach, which is typically termed *mixed parallelism* [22, 19, 39], an application consists of tasks organized in a Directed Acyclic Graph (DAG) in which each edge corresponds to a precedence relation between two tasks, implying possible data communication. Each task is itself a data-parallel task that exhibits loop parallelism, meaning that iterations can be executed, at least conceptually, in a Single Instruction Multiple Data (SIMD) fashion on various numbers of processors. (Mixed-parallel applications are also referred to as "malleable tasks with precedence constraints" in the literature.) Prime candidates for mixed-parallel implementations are scientific workflows, as seen in existing workflow systems that support mixed parallelism [46, 27, 25]. For instance, mixed-parallel applications arise for instance in image processing applications that consist of a workflow of image filters, where some of these filters can be themselves implemented as data-parallel applications [23]. See [13] for a description of the benefits of mixed parallelism and for other examples of mixed-parallel application. In this work we focus on the problem of mixed-parallel application scheduling, and like most previous works in this area we restrict our study to applications that are fully specified at the onset of their execution (e.g., static non-data-dependent workflows).

Typical platforms for executing mixed-parallel applications range from single homogeneous commodity clusters to heterogeneous multi-site grid platforms. A key question for the use of these platforms, regardless of the application at hand, is: How do applications get access to compute resources? Batch schedulers are the most common resource management systems used in production today [1, 2, 3, 4, 5]. Batch schedulers accept requests for compute resources, place these requests in queues, and grant access to resources based on various policies [18]. The main advantage of batch schedulers is that they can be configured to maximize resource utilization. The main drawback is that there is often a disconnect between the desires of application users (low turn-around time, fairness) and the schedules computed by batch schedulers [42, 28]. In particular, it is often difficult for a user to estimate when a job may complete, sometimes motivating creative strategies to mitigate this uncertainty [35, 10]. In many production scenarios such use of batch schedulers is inappropriate, for instance when applications must execute within deadlines. Instead, users need guaranteed access to resources within fixed time-frames. Consequently, most modern batch schedulers provide *advance reservation* capabilities [1, 4, 9] or can support reservations thanks to plug-in modules [3, 33]; and several production platforms have enabled the use of reservations for years [18].

In this paper we consider the problem of scheduling mixed parallel applications with advance reservations, that is on a platform that supports advance reservations and that is subject to advance reservations from competing users. We consider two scheduling problems: minimizing turn-around time and meeting a deadline. We restrict our study to the case of a single homogeneous computing platform such as a cluster, and leave the study of heterogeneous and/or multi-cluster platforms for future work. To the best of our knowledge, no previous work has studied the scheduling of mixed-parallel applications with advance reservations.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes our application, platform, and resource management models, and gives formal problem definitions. Section 4 and Section 5 present algorithms and experimental results for both our scheduling problems. Section 6 discusses the computational complexities and execution times of our proposed algorithms. Section 7 summarizes our contributions and outlines future directions.

*Aida is partially supported by the Japan Society for the Promotion of Science under Grant-in-Aid for Scientific Research (B) 18300018.
†Casanova is partially supported by the U.S. National Science Foundation under Award #0546688.

.

2. RELATED WORK

Our work is related to ones in two distinct areas: (i) scheduling of mixed-parallel applications on dedicated systems; (ii) scheduling of task-parallel applications, and in particular task-parallel workflows, with advance reservations. Hereafter we review works in both these areas, focusing solely on works that target homogeneous platforms.

2.1 Mixed-parallel Scheduling

On the theoretical side of mixed-parallel application scheduling the main contribution is the guaranteed algorithm in [29], which was later improved in [26]. Many non-guaranteed algorithms that generate good schedules in practice have been proposed [7, 37, 40, 41]. All these algorithms use two phases: a first phase in which the algorithm decides how many processors should be allocated to each task; and a second phase in which tasks are scheduled in time and space. Most notable is the CPA (Critical Path and Area-based scheduling) algorithm [37], which has low computational complexity and was shown to lead to good results when compared to its competitors. One drawback of CPA is that, for some application and platform characteristics, it may result in allocations that are too large and hinder task parallelism. The MCPA algorithm addresses this drawback in the case of layered task graphs [7]. In the general case, this drawback was also addressed in [34] via a more stringent stopping criterion for CPA's first phase, thus better limiting task allocations.

In addition to the above two-step algorithms, one-step algorithms have also been proposed [8, 47]. In particular, the iCASLB algorithm was shown to lead to better performance than some two-step algorithms, including CPA, while maintaining reasonable complexity [47]. This algorithm performs allocation and mapping simultaneously by iteratively increasing the allocations of tasks on the critical path, with a look-ahead mechanism to avoid being trapped in local minima and a backfilling approach to improve the schedule.

In this work we use CPA-computed schedules as bases for scheduling mixed-parallel applications in the presence of advance reservations.

2.2 Scheduling with Reservation

One can classify works that have studied scheduling in advance reservation scenarios in two cases, depending on whether they target independent jobs or a single job that consists of dependent sub-jobs. In the first case, the goal is to find for each job a reservation that allows that job to meet a particular deadline [17, 24, 12, 11]. In the second case, reservations for sub-jobs must be made in a coordinated manner to ensure that the last sub-job meets a global application deadline [31, 48, 15]. Our work falls in this second case, but considers sub-jobs that are themselves data-parallel applications, which to the best of our knowledge has not been studied before.

3. MODELS AND PROBLEM STATEMENT

We use simulation to evaluate our algorithms because it allows for repeatable results (in real system the workload is dynamic), because it is cheap (we do not need to burn enormous amounts of allocated CPU hours on real-world parallel platforms), and because it is fast (we can explore a wide range of scenarios and run statistically significant numbers of experiments). We outline hereafter the models implemented in our simulations and then defined our target scheduling problems.

3.1 Application and Platform Model

We model an application as a Directed Acyclic Graph (DAG) in which each vertex represents a data-parallel task, and each edge represents a precedence relation between tasks. We consider the execution of such an application on a homogeneous distributed-memory parallel computing platform, e.g., a homogeneous cluster, that comprises p processors. We use the term "processor" to refer to an individually schedulable unit: a user can ask the batch scheduler for a given number of processors starting at some fixed date and for a given duration. With this terminology, a "processor" may in fact be a physical compute node that is a multi-processor and/or multi-core computer.

Table 1: Application model parameter values. Default values when keeping a parameter fixed are shown in boldface.

Parameter	Values
Number of tasks	10, 25, **50**, 75, 100
α	.05, .10, .15, **.20**
width	.1, .2, .3, .4, **.5**, .6, .7, .8, .9
density	.1, .2, .3, .4, **.5**, .6, .7, .8, .9
regularity	.1, .2, .3, .4, **.5**, .6, .7, .8, .9
jump	**1**, 2, 3, 4

More formally, we model an application as a DAG (V,E), where V is a set of vertices, or *tasks*, $t_i, i = 1,\ldots,n$, and E is a set of directed edges. If there is an edge from task t_i to task t_j, then task t_j cannot begin executing until task t_i has completed. We call task t_j a successor of task t_i, which is a predecessor of task t_j. Without loss of generality, we assume that the DAG has a single entry task and a single exit task. In addition to the number of tasks, we use 4 parameters used in previous work to define the shape of the DAG: width, regularity, density, and jumps [34]. 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 low value means that levels contain very dissimilar numbers of tasks, while a high 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 low value leading to few edges and a large value leading to many edges. These three parameters take values between 0 and 1. Finally, we add random "jumps edges" that go from level l to level $l + jump$, for $jump = 1,2,3,4$. The case $jump = 1$ corresponds to no jumping over any level, resulting in a so-called layered DAG [7]. We refer the reader to the DAG generation program and its documentation for more details [14].

While the above generates the DAG's structure, we still have to model the execution time of tasks. We assume that each task t_i is a data-parallel task that can be executed on any number of processors between 1 and p. Like most previous works we model task execution time based on Amdahl's law: a fraction α_i of the task execution is not parallelizable. Each task execution time is then fully specified simply by its sequential execution time, T_i, and by α_i. In our simulations we pick values for α_i randomly selected between 0 and a value α using a uniform probability distribution. We pick T_i randomly between 1 minute and 10 hours. The parameters of our application models and the ranges of values we use in our experiments are shown in Table 1, which also indicates default values whenever we fix a parameter.

We do not model any communication network or any data transfer between application tasks, with the following rationale. Due to the presence of advanced reservations different sets of processors are unavailable at different times. It would be egregiously inefficient to schedule the application across a continuous period of time, i.e., with a single reservation during which all application tasks can

execute. Indeed, such a reservation would have a late start time due to other reservations, or would be limited to using only a few processors. We conclude that it is better to schedule the application using multiple reservations and we simply use one reservation per application task. In this multiple reservation scenario, a task could complete well ahead of the start time of one of its successors, thus precluding the use of network communication between the two tasks. One then must use disk I/O and implement task communication via files. The associated overhead is comprised in our model of task execution times (e.g., as part of α_i).

Finally, note that the above model assumes that we have perfect knowledge of the data-parallel task execution times for various numbers of processors. This is a common assumption in the area of mixed-parallel application scheduling: this knowledge can be gathered via benchmarking or via precise performance models. If, however, only imprecise knowledge is available then reservations would be made using pessimistic estimates of task execution times (which most users typically employ for batch job submissions [32]). More pessimistic estimates lead to task reservations later in the future, due to fewer opportunities to fill small "holes" early in the reservation schedule, and thus to longer application execution time. We leave a study of the impact of pessimistic estimates outside the scope of this paper. However, we do not anticipate that the results in this paper would be changed since all algorithms should be impacted similarly by pessimistic estimates.

3.2 Advance Reservation Model

We need to model (i) the existing reservations in the system at the time of application scheduling; and (ii) the way in which one interacts with the reservation system to schedule the application.

3.2.1 Modeling Reservation Schedules

Advanced reservation is a key feature of batch schedulers in production today. Unfortunately, little information is publicly available regarding actual *reservation schedules*, i.e., the list of ongoing and future resource reservations at a given time, where each reservation is defined as a start time, a finish time, and a number of processors. Ideally, one would obtain and analyze logs of batch schedulers running for long periods of time with detailed information both about jobs submitted to batch queues and about jobs that used advance reservations. While such logs are available for batch jobs, to the best of our knowledge there are only few public sources of logs that provide information on advance reservation jobs [6]. Therefore, very few works use real-world advance reservation logs (we are only aware of [30]), and reservation schedule modeling is an open issue.

In the face of this lack of data, the approach used in previous work is to modify available logs of batch jobs by randomly tagging some fraction of the jobs as "reserved" and removing all other jobs, thus resulting in a reservation schedule [44, 45]. We use this approach with four logs from the Parallel Workloads Archive [36] for four parallel platforms (see Table 2). For each log we select a fraction ϕ of the jobs from the log using a uniform probability distribution. In our experiments we use $\phi = 0.1$, 0.2, and 0.5, to study more or less sparse reservation schedules. We then pick a random time in the log, T, at which our application scheduling decisions take place. One problem with this method is that the resulting reservation schedule is stationary: the average number of reservations per time unit is approximately constant throughout time. Instead, one would expect that, starting at time T, the number of reservations decreases with time. Accordingly, we modify the reservation schedule using one of the three following methods:

- ***linear*** **–** We add and remove reservations in the reservation schedule at time T so that the average number of reservation jobs *per day* decreases approximately linearly throughout time (and so that no reservations are present after time $T + 7$ days).
- ***expo*** **–** Same as above, but leading to an approximately exponential decrease.
- ***real*** **–** We remove reservations in the reservation schedule at time T for jobs submitted to the batch queue after time T.

By using these three methods we not only increase the chance that at least one of the methods is more representative of the real world than those used in previous work, but more importantly we can quantify the sensitivity of our results to changes in the reservation schedule patterns.

In a preliminary attempt at validating the above approach we have obtained reservation logs for the Grid'5000 platform [21, 6]. These logs span a period of 2.5 years. We compute two simple metrics for the Grid'5000 log and our four supercomputer logs: (i) job execution times and (ii) times between job/reservation submission and job/reservation start times. For each log we compute averages and coefficient of variations for these metrics. The goal is to find whether these metrics have significantly different values for the Grid'5000 log. Results are shown in Table 3. We see that coefficients of variation are low, under 4%, and that average metric values for Grid'5000 are comparable to those for the four other logs. We also compute correlations between reservation schedules (i.e., number of reserved processors throughout time starting at random points in the log) from the Grid'5000 log and for synthetic reservation schedules generated from our 4 other logs using the *linear*, *expo*, and *real* methods described above for three values of ϕ. On average, the *linear*, *expo* and *real* methods lead to correlation coefficients of 0.27, 0.54, and 0.44, respectively. This seems to indicate that the *expo* method leads to reservation schedules that are the most correlated to those of the real-world Grid'5000 reservation schedule. However, we found that the *real* method leads to better correlations than the *expo* methods when used for some of our supercomputer logs.

A fully validated reservation schedule model is outside the scope of this paper (and would require more than one real-world data set!). We believe that our method for generating synthetic reservation schedules is reasonable, at least for our purpose. And indeed, as seen later in the paper, we do not find significant discrepancies between simulation results obtained using our (plentiful) synthetic data or our (scarce) real-world data.

3.2.2 Modeling the Reservation Process

To model the interaction between the application scheduler and the batch scheduler we make two simplifying assumptions. First, we assume that application scheduling and submission of reservation requests are instantaneous. Therefore, the reservation schedule does not change while the application is being scheduled. Second, we assume that the application scheduler has full knowledge of the reservation schedule. Some schedulers do expose this information as part of their interface (e.g., PBSpro [4], Maui [3], Plus [33]), but system administrators may not be willing to enable this feature. In this case, the application schedule would have to be determined via (a bounded number of) trial-and-error reservation requests for each application task.

3.3 Scheduling Problems

We consider two different scheduling problems, instantiated for a mixed-parallel application on a homogeneous cluster with p pro-

Table 2: Four batch job logs used for simulation experiments.

Name	Institution	Year	Duration	Architecture	#CPUs	Avg. Utilization
CTC_SP2	CTC	Jun/1996-May/1997	11 [mon]	IBM SP2	430	65.8 [%]
OSC_Cluster	OSC	Jan/2000-Nov/2001	22	linux cluster	57	38.5
SDSC_BLUE	SDSC	Apr/2000-Jan/2003	32	IBM SP	1152	75.7
SDSC_DS*	SDSC	Mar/2004-Apr/2005	13	IBM eServer p690	224	27.3

$*$ only for jobs submitted to partition 3.

Table 3: Statistics for the Grid'5000 reservation log and four non-reservation batch logs.

Log	Avg. job exec. time	CV job exec. time	Avg. time to exec.	CV time to exec.
Grid'5000	1.84 [hour]	3.54 [%]	3.24 [hour]	2.52 [%]
CTC_SP2	3.20	1.41	7.49	0.61
OSC_Cluster	9.33	2.84	3.02	1.63
SDSC_BLUE	1.18	0.77	8.90	0.69
SDSC_DS*	1.52	2.75	4.41	2.48

cessors and with a given reservation schedule at application scheduling time:

1. RESSCHED: compute a schedule that minimizes application turn-around-time, i.e., the time between schedule computation and application completion.

2. RESSCHEDDL: determine whether the application can complete before a given deadline K (and produce the schedule if the answer is "yes").

In the next two sections we discuss the complexity of both problems, propose algorithms to solve them, and evaluate these algorithms in simulation.

4. MINIMIZING TURN-AROUND TIME

4.1 Computational Complexity

In [16] the authors study the problem of application turn-around time minimization for independent sequential tasks in the presence of future reservations, which they term RESASCHEDULING. This problem is a simpler special case of problem RESSCHED due to the fact that tasks are independent and sequential. The authors prove that RESASCHEDULING is strongly NP-hard via reduction to 3-PARTITION [20]. Furthermore, they prove that the problem is inapproximable in the general case. We conclude that RESSCHED is NP-hard, and inapproximable. They describe guaranteed algorithms for two special cases. The first special case requires that the number of reserved processors be monotonically decreasing with time, which does not hold in practice. The second special case requires that a subset of the processors be un-reservable, which may be desirable in practice. However, it is not known whether the performance guarantee holds when there are dependencies among tasks and when tasks are data-parallel. Furthermore, mixed-parallel application scheduling with no reservations is also NP-hard. Guaranteed algorithms are available [29, 26] but here it is not known whether these guarantees hold in the presence of reservations.

It is very likely that the aforementioned performance guarantee do not hold in our case. We leave the development of (most likely elusive) new guaranteed algorithms for solving RESSCHED outside the scope of this paper. Instead, we resort to developing non-guaranteed heuristics.

4.2 Algorithms

Perhaps the most natural approach for scheduling a task-parallel DAG with advance reservations is to adapt popular list-scheduling heuristics so that they account for existing advance reservations [31, 48, 15]. In the case of problem RESSCHED, the added challenge is that the application is mixed-parallel.

A well-known and simple algorithm for scheduling mixed-parallel applications is CPA [38]. This algorithm uses two phases: (i) an *allocation phase* in which it decides how many processors should be allocated to each task; and (ii) a *mapping phase* in which tasks are scheduled in decreasing order of their bottom levels, which is a standard list scheduling approach. Recall that the bottom level of a task is the maximum length of the paths from this task to the end task in the DAG, expressed as a sum of task execution times, and easily computed recursively. In its first phase, CPA starts by allocating one processor to each task. Then, CPA iteratively increases the allocation of one task by one processor. This task is chosen at each step as the task on the critical path whose execution time would be reduced the most (relatively) when given an extra processor. The algorithm uses a stopping criterion for the allocation phase, trying to achieve the best compromise between the length of the critical path and the average processor utilization. This criterion was later modified in [34], where it is shown empirically that the modified criterion leads to better schedule (lower makespan, higher efficiency). When we refer to CPA hereafter we mean this improved version of the algorithm.

A natural approach would then be to adapt the CPA algorithm to an advance reservation scenario. A major difficulty here is that task allocations are computed disregarding the reservation schedule, meaning that task start times could be delayed significantly. Instead, we propose an algorithm that proceeds in the following two simple phases:

1. Compute a bottom level value for each task and sort the tasks in decreasing order of their bottom levels;

2. For each task, in order, determine the best feasible task allocation, i.e., a number of processors and a start time, that allows the task to achieve the earliest completion time, accounting for competing reservations.

The above, albeit simple, makes it possible to increase or reduce task allocations while considering the time at which the task would execute, and thus also considering existing advance reservations at that time. Two questions arise:

1. How should bottom levels be computed?

2. Should task allocations be bounded?

Let us discuss these two questions and propose several approaches, whose combinations result in a number of candidate algorithms for solving RESSCHED.

Computing task bottom levels requires that all task execution times be known. The execution time of a task must be computed for a given number of processors. We consider four options:

1. **BL_1 –** task execution times are computed assuming that each task is allocated a single processor.
2. **BL_ALL –** Task execution times are computed assuming that each task is allocated p processors.
3. **BL_CPA –** Task execution times are computed using the task allocations computed by the first phase of the CPA algorithm when considering that up to p processors, i.e., all processors, can be used.
4. **BL_CPAR –** Task execution times are computed using the task allocations computed by the first phase of the CPA algorithm when considering that up to $q \leqslant p$ processors are available. q is the historical average number of available processors for the platform, computed based on the past reservation schedule.

Our rationale for the last two options above is that, as we will see below, one of our goals is to achieve a schedule that is inspired by a CPA-computed schedule. Therefore, one may surmise that using CPA-computed allocations to compute the bottom-level may ultimately lead to better schedules (although there is no guarantee). The rationale for the BL_CPAR approach is that the CPA-computed allocations will in the end be bounded by some maximum number of *available* processors. A coarse approximation for this number is the historical average number of available processors (in practice the number of available processors is likely non-stationary).

For phase two, we consider three ways in which task allocations can be bounded:

1. **BD_ALL –** Task allocations are simply bounded by p, meaning that all options are considered for each task when picking the allocation that achieves the earliest completion time.
2. **BD_CPA –** Task allocations are bounded by the CPA-computed allocations assuming that p processors are available.
3. **BD_CPAR –** Task allocations are bounded by the CPA-computed allocations assuming that q processors are available, where q is the historical average number of available processors.

One of the insights behind CPA is that it bounds task allocations to foster task parallelism in addition to enabling data parallelism. Therefore, bounding task allocations based on CPA-computed allocations should be beneficial both in terms of application turn-around-time and in terms of parallel efficiency.

The above defines $4 \times 3 = 12$ algorithms for solving the RESSCHED problem, each named BL_x_BD_y, where $x \in \{1, \text{ALL}, \text{CPA}, \text{CPAR}\}$ and $y \in \{\text{ALL}, \text{CPA}, \text{CPAR}\}$. Note that if the reservation schedule is empty, then the BL_CPA_BD_CPA algorithm is simply the CPA algorithm.

4.3 Experimental Evaluation

4.3.1 Computing Bottom Levels

We first determine which of the techniques for computing task bottom levels is the most effective. We compare the algorithms in simulation, with a simulator that implements the models described in Section 3. We run simulations for applications generated with the parameter values in Table 1 by fixing five of the six parameters to their default values but for one parameter. For each of the resulting $5+4+9+9+9+4 = 40$ applications specifications we generate 20 sample random application instances. We use the job submission logs described in Table 2. For each log we tag ϕ =0.1, 0.2, or 0.5 of the jobs as reservations, and we then use the *linear*, *expo*, and *real* methods described in Section 3.2.1, leading to $4 \times 3 \times 3 = 36$ reservation schedule specifications. For each specification we generate 50 random reservation schedule instances as follows. We pick 10 different start times in each trace, i.e., the time at which the application is being scheduled. For each start time we generate 5 sample random taggings of jobs as reservation jobs. Overall, we have $40 \times 36 = 1,440$ experimental scenarios, and for each scenario we have $20 \times 50 = 1,000$ random instances. In all that follows we discuss average results over the $1,000$ random instances for each scenario and omit the word "average" for brevity.

For each experimental scenario we schedule the application using all 12 algorithms described in the previous section. For each experimental scenario and for each allocation bounding method, we compute the improvement of the turn-around-time when using the BL_ALL, BL_CPA, and BL_CPAR bottom-level computation method relative to using the BL_1 method.

Over all our experiments the relative turn-around-time improvement over using the BL_1 method is between -3.46% and $+5.69\%$, with negative values denoting that the BL_1 methods leads to better performance. This result quantifies the (moderate) sensitivity of application scheduling to bottom-level computations. Looking at the results in more details, we found that the BL_CPA and the BL_CPAR methods together lead to the best turn-around-time in 78.4% of the cases. The BL_CPAR method is better than the BL_CPA method for more than two thirds of these cases. When it is worse than BL_CPA, its relative performance when compared to the BL_1 method is better than that of BL_CPA by at most 0.3 points. The BL_1 method is best in 13.7% of the cases, and the BL_ALL method in 7.9% of the cases. Finally, we found no particular trends regarding the 6 parameters defined in Table 1, or with the method used to generate reservations (*linear*, *expo*, or *real*).

We conclude that BL_CPAR is the best method (but only marginally better than BL_CPA). It is important to note that this result holds *regardless* of the method used for bounding allocations. We hypothesize that this is because task allocations in the final schedule most resemble those produced by the CPA algorithm when using the number of available processors (rather than, e.g., 1-processor allocations). This intuition is partially confirmed via a more thorough examination of our results. Indeed, we found not only that the BL_1 method leads to the worst results, but also that the gap between the BL_1 method and the other three methods decreases when the total number of processors in the platform decreases or when the number of reservations increases.

The above observations are confirmed in similar experiments conducted with 50 reservation schedules extracted from the Grid'5000 log at 50 random start times. For the remainder of this paper we exclusively use the BL_CPAR method for computing task bottom levels.

4.3.2 Bounding Allocations

Using the same experimental methodology as in the previous section we schedule an application instance for a given reservation schedule instance using the three allocation bounding methods: BD_ALL, BD_CPA, BD_CPAR. We also use another method that arbitrarily bounds allocations to half of the total number of processors: BD_HALF. We include this method to evaluate whether

sophisticated allocation bounding based on the CPA algorithm is truly necessary or whether a simple bounding technique would be sufficient.

We use two metrics to evaluate the schedule: (i) the application turn-around-time; and (ii) the CPU-hour consumption. While the objective when solving RESSCHED is to minimize application turn-around-time, it is still interesting to observe how efficient the schedule is. Ideally, one would like to find an algorithm that leads to the best turn-around-times with the lowest resource consumption.

Table 4 shows a summary of the results over our 1,440 experimental scenarios for each algorithm in terms of turn-around-time and CPU-hour usage. For each metric we show the average degradation from best. For an experimental scenario, the degradation from best for an algorithm is computed as the average relative difference between the metric for this algorithm and the metric for best-performing algorithm. We then take the average over all scenarios. A value close to 0 means that the algorithm is on average close to the best performing algorithm. For each metric we also show the number of times the algorithm is best across all the scenarios. Note that the total number of "wins" is slightly higher than 1,440 due to ties.

The average degradation from best shows that the BD_CPAR algorithm is the best algorithm both in terms of turn-around-time *and* in terms of CPU-hour usage. The BD_CPA algorithm is a close runner-up in terms of turn-around-time but uses more CPU-hours. This demonstrates that using the historical average number of available processors to compute CPA allocations leads to some benefit. The BD_ALL algorithm exhibits the worst performance overall. Since it does not bound allocations at all it harms task parallelism and, in turn, turn-around time. Also, this algorithm wastes compute resources due to the diminishing returns of Amdahl's law and its use of large allocations. The BD_HALF algorithm also leads to poor results, showing that arbitrary bounding of task allocation is likely not a good approach and that application characteristics must be taken into account for the bounding. Interestingly, BD_CPA has more "wins" than BD_CPAR but these wins are very slight. By contrast, when BD_CPA loses to BD_CPAR it loses by a larger margin because it uses allocations that are too large relatively to the number of processors that are actually available.

While several of the parameters defining our experimental scenarios do not have any effect on our results we nevertheless observe a few trends:

- When DAG width is low the BD_ALL algorithm leads to the best turn-around-times. In this case the DAG is essentially a chain and the best turn-around-time is achieved when using allocations that are as large as possible. In fact, all 36 wins by BD_ALL in Table 4 are for a DAG width of 0.1 (and in these cases BD_ALL leads to turn-around-times at most 0.5% shorter than those obtained by BD_CPAR). For large DAG widths, the gap between the BD_ALL and BD_HALF algorithms and the BD_CPA and BD_CPAR algorithm increases, due to these last two algorithms' ability to capitalize on task parallelism.

- As the number of competing reservations in the reservation schedule increases the gap between the BD_ALL algorithm and the other algorithms decreases (but their ranking is preserved). This is easily explained: with fewer available processors the allocations computed by BD_ALL are not as large and thus do not preclude task parallelism as drastically. For the same reason, the gap between the BD_ALL algorithm and the other algorithms decreases for smaller platforms and for more utilized platforms.

Table 5 shows a summary of the results obtained by conducting similar experiments for our 40 application scenarios, but with 50 reservation schedules extracted at 50 random times from the Grid'5000 log. We see that the results are similar to those in Table 4 for our synthetic reservation schedules (with actually more turn-around time "wins" for BD_CPAR than for BD_CPA).

We conclude that among our candidate algorithms for solving RESSCHED the best approach, which in addition leads to low resource consumption, is to use the BL_CPAR method to compute task bottom-levels and the BD_CPAR method to bound task allocations. Note however that the improvement over the BL_CPA and BD_CPA methods are not drastic.

5. MEETING A DEADLINE

5.1 Computational Complexity

RESSCHEDDL is essentially the decision problem associated to the RESSCHED optimization problem, and we have seen the NP-hardness of RESSCHED in Section 4.1. Here again we do not develop guaranteed algorithms but instead propose several non-guaranteed heuristics.

5.2 Algorithms

The most natural idea is to use the same scheduling approach as for solving RESSCHED, but "backwards", i.e., scheduling tasks in order of *increasing* bottom-levels and scheduling them backward in time starting from the deadline, K. We use the BL_CPAR method for computing bottom-levels because it proved the best for solving RESSCHED. For each task t_i to be scheduled, its successors have already been scheduled (because they have lower bottom-levels). Therefore, one must schedule t_i so that it finishes no later than the minimum of the start times of its successors (or than K if t_i is the first task being scheduled). Each task scheduling decision becomes a deadline scheduling decision, where each task has a different deadline. This decision boils down to picking a number of processors for task t_i and find a reservation (for this number of processors and for the corresponding task execution time) that is as late as possible. The reservation should be as late as possible to leave as much time as possible between the time at which the application is being scheduled (which we term "now") and t_i's start time, thus making it easier to schedule all tasks with a higher bottom-level than task t_i.

In the above scheme, for each task scheduling decision there are typically several possible numbers of processors. Each of these numbers of processors would lead to a different start time for task t_i. Therefore we must pick one of several <number of processors, start time> pairs. The question is: which one should be picked?

5.2.1 Aggressive Algorithms

We can use the same three approaches as for bounding task allocations when solving the RESSCHED problem: BD_ALL, BD_CPA, and BD_CPAR. We then obtain three algorithms (whose names start with "DL_" as in DeadLine):

1. **DL_BD_ALL –** For each task, in order of increasing bottom-levels, pick the <number of processors, start time> pair that leads to the latest task start time.

2. **DL_BD_CPA –** Same as DL_BD_ALL, but bound the the number of processors by the CPA-computed allocation assuming that $q = p$ processors are available.

Table 4: Results for turn-around time minimization (with synthetic reservation schedules).

Algorithm	Turn-around-time		CPU-hours	
	Avg. deg. from best	Number of wins	Avg. deg. from best	Number of wins
BD_ALL	33.75 [%]	36	42.48 [%]	0
BD_HALF	28.38	3	37.83	1
BD_CPA	0.29	1026	0.75	6
BD_CPAR	0.21	386	0.00	1,434

Table 5: Results for turn-around time minimization (with Grid'5000 reservation schedules).

Algorithm	Turn-around-time		CPU-hours	
	Avg. deg. from best	Number of wins	Avg. deg. from best	Number of wins
BD_ALL	34.32 [%]	0	43.08 [%]	0
BD_HALF	30.43	9	29.17	0
BD_CPA	0.19	9	0.82	0
BD_CPAR	0.15	30	0.00	40

3. **DL_BD_CPAR –** Same as DL_BD_CPA, but with q equal to the historical average number of available processors.

Based on the results obtained with similar approaches for solving RESSCHED, we expect that algorithm DL_BD_CPAR would be the best of the three. We term these three algorithms "aggressive" because they do not attempt to use fewer processors when the deadline is looser.

5.2.2 Resource Conservative Algorithms

One problem with the previous three algorithms is that they do not adapt the number of processors they use to the tightness of the deadline. If the deadline is loose, then the application could be scheduled using very few processors per task and still meet the deadline. As a result the number of CPU-hours spent would be reduced, a useful feature since users typically have a limited CPU-hour budget. The goal is then to pick a <number of processors, start time> pair with the lowest number of processors that will not preclude meeting the deadline. The challenge is that it is difficult to reason about which task start time is late enough to ensure (with high probability) that the deadline will be met.

We propose one approach for addressing this challenge. Our insight, when scheduling a task t_i, is to use the CPA algorithm to compute a full schedule for all the application tasks that have not been scheduled yet. This full schedule, computed for some number of processors q, will then produce a particular start time for task t_i, say S_i. Provided that q is approximately equal to the number of processors available between "now" and t_i's deadline, S_i provides a good guideline for picking a start time for task t_i. A start time much earlier than S_i would most likely make it impossible to meet the deadline, while a start time much later than S_i would likely waste CPU-hours. Our basic strategy when scheduling task t_i is to first compute S_i and then pick the <number of processors, start time> pair with the lowest number of processors (to save resources) so that the start time is later than S_i (to still meet the deadline).

The above strategy results in two algorithms, each for a different values of q, as for DL_BD_CPA and DL_BD_CPAR above. The two algorithms are named DL_RC_CPA and DL_RC_CPAR, with RC standing for "Resource Conservative":

1. **DL_RC_CPA –** For each task, in order of increasing bottom-level, pick the <number of processors, start time> pair that leads to the earliest task start time that is larger than the CPA-computed start time assuming that $q = p$ processors are available.
2. **DL_RC_CPAR –** Same as DL_RC_CPA but with q equal to the historical average number of available processors.

Note that if the above algorithms fail to find a <number of processors, start time> with a start time after the CPA-computed start time, then they revert to an aggressive mode and choose the pair that has the latest start-time, in an attempt to get "back on track".

5.3 Experimental Evaluation

We use two performance metrics. First, for each problem instance we determine the tightest deadline that each algorithm can achieve, with an earlier such deadline denoting better performance. This tightest deadline is determined via binary search. Second, we compute the number of CPU-hours consumed by the schedule produced by each algorithm for a deadline that is 50% as large as the latest tightest deadline across all the algorithms. This second deadline quantifies the resource consumption in a "loose deadline" scenario. We do not conduct experiments for absolute deadline values because whether a deadline is loose or tight strongly depends on the problem instance, i.e., on the application configuration and the reservation schedule.

We compare our five algorithms in simulation using the same experimental methodology as in Section 4.3. Due to the enormous number of experiments and to the fact that searching for the tightest deadline is time consuming, we show results obtained using only the SDSC_BLUE log out of our for logs, thus corresponding to 360 out of our total $1,440$ synthetic experimental scenarios, with $1,000$ random RESSCHEDDL instances for each. We also show results obtained for 50 reservation schedules extracted from the Grid'5000 logs at random start times.

Table 6 shows a summary of the results using percentage average degradation from best as in Section 4.3.2. For each metric we show four columns of results, one showing average results for each value of ϕ and one for the Grid'5000 reservation schedules. Recall that a higher ϕ means a less sparse reservation schedule. The first observation is that in all cases the resource conservative algorithms, DL_RC_CPA and DL_RC_CPAR, lead to dramatically lower resource consumption than the aggressive algorithms when given loose deadlines. The DL_BD_ALL aggressive algorithm leads to resource consumption that is orders of magnitude higher than the other algorithms due to its use of overly large allocations. As expected also, the DL_BD_CPAR algorithm uses slightly fewer resources than the DL_BD_CPA algorithm because of its use of smaller allocations.

In terms of deadline tightness the DL_BD_ALL algorithm leads

Table 6: Results for algorithms that attempt to meet a deadline

Algorithm	Tightest deadline (Average % degradation from best)				CPU-hours for loose deadline (Average % degradation from best)			
	$\phi = 0.1$	$\phi = 0.2$	$\phi = 0.5$	Grid'5000	$\phi = 0.1$	$\phi = 0.2$	$\phi = 0.5$	Grid'5000
DL_BD_ALL	178.43	175.58	188.33	227.03	3556.70	3486.30	3769.20	2006.30
DL_BD_CPA	6.11	6.16	6.26	8.00	252.30	251.36	275.05	185.58
DL_BD_CPAR	6.52	6.44	6.91	8.38	231.01	236.97	243.60	179.35
DL_RC_CPA	13.17	13.27	17.36	19.51	6.39	6.80	7.98	2.15
DL_RC_CPAR	4.12	4.27	8.26	15.13	0.16	0.15	0.16	0.09

to very poor performance, again due to its use of overly large allocations that hinder task parallelism. More interestingly, for $\phi = 0.1$ and $\phi = 0.2$, DL_RC_CPAR leads to marginally tighter deadlines than the aggressive algorithms! This is because DL_RC_CPAR tries to match the CPA schedule as much as possible, which has the effect of both saving CPU-hours and leading to short turn-around times, and thus to possibly tighter deadlines. For $\phi = 0.5$, we see that the aggressive algorithm perform marginally better. With $\phi = 0.5$ the reservation schedule is less sparse and there are more chances for DL_RC_CPAR to be "caught in a bind", for instance after a large fraction of the DAG has been scheduled. With such reservation schedules, it is very difficult to match the CPA schedule as much as possible, and attempting to match it leads to missing tight deadlines. Instead, the aggressive algorithms are deadline conservative since they use more resources whenever possible. They are therefore less susceptible to missing a tight deadline. This phenomenon is amplified for the DL_RC_CPA, which leads to poorer results than DL_RC_CPAR and than the aggressive algorithms: it misses tight deadlines because it overestimates the number of available processors.

Our results indicate no particular trends with respect to most of the 6 application parameters defined in Table 1, or with the method used to generate reservations (*linear*, *expo*, or *real*). The only exception is the number of tasks. We witness the same trend when the number of task increases as when ϕ increases, meaning that the resource conservative algorithms do not achieve as tight deadlines as the aggressive algorithms. Here again it is because with more tasks in the application there are more opportunities for the resource conservative algorithms to be left in a bind when attempting to meet a tight deadline.

Finally, the results for the Grid'5000 reservation schedules are consistent with those for the synthetic reservation schedules, although the advantage of the resource-conservative algorithms are marginally lower in terms of deadline tightness and marginally higher in terms of CPU-hours.

Our conclusion so far is that the BD_RC_CPAR algorithm is the best choice among the five algorithms. In all cases BD_RC_CPAR leads to resource consumption that is lower than that of BD_RC_-CPA and orders of magnitude lower than that of the aggressive algorithms. In low reservation load conditions (i.e., $\phi = 0.1$ or $\phi = 0.2$) it leads to tightest deadlines tighter than or comparable to the tightest deadlines achieved by its competitors. In high reservation load conditions (i.e., $\phi = 0.5$), and for the Grid'5000 reservation schedules, it leads to tightest deadlines that are looser than the aggressive algorithms. We investigate this phenomenon further in the next section.

5.4 Hybrid Algorithm

Although the results in the previous section indicate that our resource conservation approach is effective, it fails to outperform the aggressive approach when the reservation schedule is less sparse. This observation is most pronounced for simulations conducted with our real-world reservation schedule. This raises an important concern, which we address in this section. We consider only the DL_BD_CPA aggressive algorithm and the DL_RC_CPAR resource conservative algorithm in this section as they both have been shown to outperform other algorithms in their categories.

The reason why DL_RC_CPAR achieves markedly poorer tightest deadlines than DL_BD_CPA is that it tries to be too conservative. Indeed, by always scheduling tasks as early as possible, it can find itself lacking options for scheduling the first application tasks (i.e., those higher up in the application DAG). This happens for instance when the reservation schedule is not uniformly sparse throughout time, and in particular when it is less sparse in the nearer future. This phenomenon was not observed for reservation schedules that are overall very sparse (results for $\phi = 0.1$ and $\phi = 0.2$ in the previous section), but became observable for less sparse schedules (results for $\phi = 0.5$ and for the Grid'5000 dataset in the previous section). For instance, for the Grid'5000 dataset, the tightest deadline achieved by DL_RC_CPAR is more than twice that achieved by DL_BD_CPA in 16% of our simulation results. However, it is 2% larger for less than 25% of the cases. This means that among the reservation schedules we extracted from the Grid'5000 dataset, about 75% of them do not cause the resource conservative algorithm to be "caught in a bind". But in the remaining 25% there are many who cause it to lead to very poor tightest deadlines relatively to the aggressive algorithm.

A way to remedy the above problem is to make DL_RC_CPAR less resource conservative. As described in Section 5.2.2, for each task t_i the algorithm picks the <number of processors, start time> pair with the lowest number of processors and with a task start time $\geqslant S_i$, where S_i is the CPA-computed task start time. Instead it could pick the pair with the lowest number of processors such that the task start time is $\geqslant S_i + \lambda \times (dl_i - S_i)$, where dl_i is the time by which task t_i must complete (so that DAG precedence constraints are not violated), and λ is a tunable parameter that takes values between 0 and 1. With $\lambda = 0$, we have our original DL_RC_CPAR algorithm. With $\lambda = 1$, we have our DL_BD_CPA aggressive algorithm. We call this new hybrid algorithm DL_RC_CPAR-λ.

The main difficulty here is to pick the best λ value. As seen earlier, there are reservation schedules for which a low λ value is a good choice, and others for which a low λ value is a poor choice. The choice of λ thus depends on the reservation schedule at hand, and there is no a-priori best value. To address this difficulty we adopt a pragmatic solution. We start with $\lambda = 0$, attempt to meet the deadline, and increase λ until the specified deadline can be met, if at all. This amounts to trying to be as resource conservative as possible while still meeting the deadline, in a way that accounts for the specifics of the reservation schedule. In our current implementation we vary λ from 0 to 1 with a step size of 0.05.

Table 7 shows simulation results for our Grid'5000 dataset using the exact same methodology and metrics as those used in the

Table 7: Results for meeting a deadline for the Grid'5000 dataset, using two improved resource conservative algorithms.

Algorithm	Tightest deadline (Average % degradation from best)	CPU-hours for loose deadline (Average % degradation from best)
DL_BD_CPA	10.96	123.98
DL_RC_CPAR	55.08	1.57
DL_RC_CPAR-λ	4.73	24.46
DL_RCBD_CPAR-λ	2.57	21.65

previous section. We see the same trend between the DL_BD_CPA and DL_RC_CPAR algorithms as in the previous section. (The actual average percent degradation from best numbers are different because the set of algorithms being compared has changed.) The third row of the table shows results for the DL_RC_CPAR-λ algorithm. We see that this algorithm outperforms the aggressive DL_BD_CPA algorithm both in terms of tightest deadline and in terms of CPU-hours used for a loose deadline. On average in our experiments, the DL_RC_CPAR algorithm saves 544 CPU-hours when compared to the aggressive algorithm, while DL_RC_CPAR-λ saves 478 CPU-hours, i.e., 12.1% less. In terms of the tightest deadlines, DL_RC_CPAR-λ leads to tightest deadlines that are on average 13.6% tighter relative to those achieved by the DL_RC_-CPAR, and 4.8% tighter relative to those achieved by DL_BD_-CPA. We conclude that the DL_RC_CPAR-λ is the algorithm of choice and should be used in most practical situations. We note however, than some users may wish to keep $\lambda = 0$ to achieve the best possible CPU-hour savings, at the cost of missing tight deadlines.

We propose here a last improvement to our DL_RC_CPAR-λ algorithm. Recall from Section 5.2.2 that our resource conservative algorithms take an aggressive approach when, for a given tasks, they cannot find a <number of processors, start time> pair that has a start time after the CPA-computed start time. In this case they try to find the <number of processors, start time> pair that has the latest start time. Instead of taking this aggressive approach, one can bound the number of processors used by the task based on the CPA-computed schedule. This is the same idea as that used by our aggressive algorithm and was shown to be effective in that context. We denote by DL_RCBD_CPAR-λ the modification of our DL_RC_CPAR-λ algorithm that uses such bounding. The bounding is based on CPA-computed task allocations computed using the historical average number of available processors. The last row of Table 7 shows that this modification is effective and leads to marginal improvements both in deadline tightness and in CPU-hour consumption.

6. ALGORITHM COMPLEXITIES AND EXECUTION TIMES

6.1 Asymptotic Computational Complexities

Let V be the number of vertices in the application DAG (i.e., the number of tasks), E the number of edges in the application DAG, P the total number of processors, P' the historical average number of available processors, R the total number of existing reservations in the current reservation schedule, and R' the number of existing reservations in the reservation schedule before the application deadline, if such a deadline is specified. Based on these values we can determine the worst-case asymptotic computational complexities (or "complexities" for short) of all our algorithms.

Following the results in Section 4.3.1, all our algorithms compute task bottom-levels by first computing task allocations using the CPA algorithm for the historical average number of available processors, P'. The complexity of the allocation phase of the CPA algorithm is $O(V(V+E)P')$ according to the analysis in [37]. Once task allocations are computed, bottom-levels need to be computed. There are efficient algorithms to compute bottom levels in time $O(V+E)$, based on a topological ordering of the vertices [43]. All algorithms then sort the tasks by decreasing bottom-level, which can be done in time $O(V \log V)$. Therefore the overall complexity of the first phase of all our algorithms is $O(V(V+E)P')$.

Let us first consider the algorithms for minimizing turn-around time described in Section 4. For each task, these algorithms attempt to find the <number of processors, start time> pair that leads to the earliest task start time. For one task, the complexity of this process is obtained by multiplying the number of reservations in the reservation schedule by the number of possible numbers of processors that can be allocated to the task. Indeed, consider a single task. One may need to go through the entire reservation schedule and attempt to "fit" the task starting after the end of each existing reservation. For now, let us call N the number of possible numbers of processors that can be allocated to the task. For the first task being scheduled, the complexity is thus $O(RN)$. For the second task being scheduled the complexity is $O((R+1)N)$, due to the new reservation in the reservation schedule corresponding to the first task. Summing over all tasks, we obtain the overall complexity of the scheduling procedure as $O(VRN+V^2N)$. For the BD_ALL and BD_CPA algorithms, $N = P$, while for the BD_CPAR algorithm $N = P'$. Note that for the BD_CPA algorithm the CPA-computed allocations using the total number of processors need to be computed. Indeed, during the bottom-level computation phase, those allocations are computed using the historical average number of available processors, and are thus different. This adds $O(V(V+E)P)$ to the complexity of this algorithm. After eliminating redundant and dominated terms we obtained the overall complexities of these three algorithms as summarized in the top part of Table 8. Note that in most practical cases one would not go through the whole reservation schedule for each task (e.g., due to precedence constraints, due to pleasantly sparse reservation schedules).

The analysis for the complexity of the aggressive algorithms for meeting a deadline, described in Section 5.2.1, is similar to that of the algorithms for minimizing turn-around time, mainly replacing R, the number of existing reservations, by R', the number of existing reservations before the specified deadline. Complexities, after eliminating redundant and dominated terms, are summarized in the middle part of Table 8.

The analysis is almost identical for the resource conservative algorithms in Section 5.2.2. A small difference is that these algorithms need to compute CPA schedules using a list-scheduling heuristic. For DL_RC_CPA this is done in $O(VP)$, while for DL_-RC_CPAR this is done in $O(VP')$ [37]. A CPA schedule need to be computed each time a task is scheduled, leading to $O(V^2P)$ and $O(V^2P')$ for DL_RC_CPA and DL_RC_CPAR, respectively. The complexity of our hybrid algorithm DL_RC_CPAR-λ, described in in Section 5.4, is identical to that of DL_RC_CPAR. Indeed, DL_RC_CPAR-λ simply applies DL_RC_CPAR a finite number

Table 8: Worst-case asymptotic computational complexities of our algorithms.

Algorithm	Complexity
BD_ALL	$O(V^2P' + V^2P + VEP' + VRP)$
BD_CPA	$O(V^2P' + V^2P + VEP' + VEP + VRP)$
BD_CPAR	$O(V^2P' + VEP' + VRP')$
DL_BD_ALL	$O(V^2P' + V^2P + VEP' + VR'P)$
DL_BD_CPA	$O(V^2P' + V^2P + VEP' + VEP + VR'P)$
DL_BD_CPAR	$O(V^2P' + VEP' + VR'P')$
DL_RC_CPA	$O(V^2P' + V^2P + VEP' + VEP + VR'P)$
DL_RC_CPAR	$O(V^2P' + VEP' + VR'P')$
DL_RC_CPAR-λ	$O(V^2P' + VEP' + VR'P')$
DL_RCBD_CPAR-λ	$O(V^2P' + VEP' + VR'P')$

of times. Finally, DL_RCBD_CPAR-λ has the same complexity as DL_RC_CPAR-λ. Complexities, after eliminating redundant and dominated terms, are summarized in the bottom part of Table 8.

We conclude that all our algorithms are polynomial and have similar worst-case asymptotic computational complexities.

6.2 Execution Times

While worst-case asymptotic computational complexities are interesting, they do not provide a good sense for how fast our algorithms are in practice and for how they compare to each other. In this section we show simulation results that compare our algorithms in terms of execution times, for our own implementation of them, in C, on a 2.4GHz AMD Opteron processor.

We measured average execution times for scheduling an application given a reservation schedule, for all the algorithms discussed in this paper. Application DAGs are generated using our application model described in Section 3.1. We use the Grid'5000 reservation schedules. All parameters defining the application take the default values shown in Table 1 but for n, the number of tasks, and d, the density of edges, which we vary. Table 9 shows execution times in milliseconds as n varies, and Table 10 shows execution times in milliseconds as d varies. The results in the table show that, expectedly, the average execution time of 1,000 random instances (20 application instances $\times$ 50 reservation schedule instances) for all algorithms increases with n and d.

Algorithms to minimize turn-around times (BD_$\star$) all take very little time. The best such algorithm, BD_CPAR, takes on average under 16 milliseconds for DAGs with $n = 100$ and $d = 0.5$, and under 5 milliseconds for DAGs with $n = 50$ and $d = 0.9$. Expectedly, results are identical for aggressive algorithms that attempt to meet a deadline. Indeed, these algorithms operate in the same manner as the algorithms that minimize turn-around time, but "backwards".

The resource-conservative algorithms for meeting a deadline are more expensive than their aggressive counterparts by roughly a factor 10 to 90. This is explained by the fact that they compute many CPA schedules. Therefore, in spite of these algorithms not exhibiting higher worst-case asymptotic computation complexity (see Table 8), in practice they do lead to higher execution times. Nevertheless, although a factor 90 is significant, we still note that our best algorithm, DL_RCBD_CPAR-λ, computes schedules in under 1.5 seconds even for DAGs that have $n = 100$ tasks. We conclude that all our algorithms can be used in practice.

7. CONCLUSION

We have studied the problem of scheduling mixed-parallel workflows using advance reservations on a parallel platform that is subject to advance reservations from competing users. We have defined two scheduling problems, RESSCHED and RESSCHEDDL, depending on whether the goal is to minimize application turn-around-time or to meet a deadline, respectively. For each problem we have proposed and evaluated a set of scheduling algorithms. These algorithms were compared for both their performance (i.e., turn-around-time and tightest deadline) and their resource consumption. Some of these algorithms reuse and adapt some of the ideas developed in the CPA algorithm [37] to compute task-bottom levels, to bound task allocations, and to compute the most resource-conservative allocations. We found these algorithms to be the most effective when using a CPA-schedule computed with an approximation (albeit coarse) of the expected available number of processors. Most notably, CPA-inspired schedule can be used to drastically reduce CPU-hour consumption. Our simulator is publicly available[1].

This work can be extended in three main directions. To the best of our knowledge this paper is the first to propose scheduling algorithms for mixed-parallel applications in an advance reservation scenario. A first natural direction is to improve on the algorithms that we have proposed to solve our scheduling problems. For instance, it would be interesting to use the iCASLB algorithm [47] instead of CPA. In fact, iCASLB could perhaps be adapted directly to advance reservation scenarios. A second direction, this one on the practical side, would be to explore how the algorithms can be implement in real-world systems by removing some of our assumptions. Prime candidates for removal are our assumption that while the application is being scheduled the reservation schedule does not change, and our assumption that the scheduling algorithm has full knowledge of the reservation schedule. A broader question of course is to consider platforms beyond a single homogeneous cluster, such as heterogeneous multi-grid platforms. It would be interesting to see how mixed-scheduling algorithms for heterogeneous platforms [34] could be adapted to consider advance reservations.

acknowledgements

The authors wish to thank Dr. Franck Cappello, the Grid'5000 team, and the Grid Workloads Archive team for providing the Grid'5000 reservation logs. The batch job logs used in this paper, CTC_SP2, OSC_Cluster, SDSC_BLUE and SDSC_DS, are graciously provided by Dr. Dan Dwyer, Supercluster.org's HPC Wordload/Resource Trace Repository, Dr. Travis Earheart and Dr. Nancy Wilkins-Diehr, and Dr. Victor Hazelwood, respectively. Finally, the author wish to thank Dr. Frédéric Suter for the DAG generation program.

[1] `http://navet.ics.hawaii.edu/~casanova/software/res_simulator.tar.gz`

Table 9: Average algorithm execution times as n varies.

Algorithm	Execution time [ms]				
	n=10	n=25	n=50	n=75	n=100
BD_ALL	0.531	1.981	5.681	12.310	20.264
BD_CPA	0.217	1.206	4.059	10.025	17.244
BD_CPAR	0.201	1.112	3.753	9.289	15.920
DL_BD_ALL	0.524	1.946	5.585	12.112	19.937
DL_BD_CPA	0.387	2.197	7.607	18.815	32.588
DL_BD_CPAR	0.205	1.117	3.764	9.272	15.977
DL_RC_CPA	2.711	30.084	200.079	732.413	1690.289
DL_RC_CPAR	2.303	24.813	166.258	627.961	1474.890
DL_RC_CPAR-λ	2.347	24.810	166.360	629.305	1476.448
DL_RCBD_CPAR-λ	2.351	24.881	166.140	628.664	1474.986

Table 10: Average algorithm execution times as d varies.

Algorithm	Execution time [ms]								
	d=0.1	d=0.2	d=0.3	d=0.4	d=0.5	d=0.6	d=0.7	d=0.8	d=0.9
BD_ALL	4.971	5.119	5.315	5.507	5.662	5.792	6.019	6.090	6.200
BD_CPA	3.017	3.255	3.547	3.825	4.053	4.204	4.491	4.605	4.734
BD_CPAR	2.797	3.013	3.286	3.535	3.771	3.885	4.149	4.267	4.361
DL_BD_ALL	4.878	5.011	5.210	5.398	5.571	5.690	5.918	6.008	6.104
DL_BD_CPA	5.614	6.095	6.616	7.161	7.596	7.857	8.431	8.622	8.852
DL_BD_CPAR	2.814	3.046	3.292	3.553	3.765	3.893	4.156	4.264	4.384
DL_RC_CPA	147.555	159.882	174.086	188.301	199.822	206.957	221.579	226.611	232.671
DL_RC_CPAR	123.758	134.074	145.794	156.901	166.037	172.068	184.152	188.308	193.274
DL_RC_CPAR-λ	123.758	134.078	145.782	156.937	166.084	172.348	184.114	188.304	193.341
DL_RCBD_CPAR-λ	124.357	134.602	145.725	156.896	166.066	172.121	184.443	188.329	193.234

8. REFERENCES

[1] Grid Engine. http://gridengine.sunsource.net.

[2] LSF. http://www.platform.com/Products/Platform.LSF.Family/.

[3] Maui cluster scheduler. http://www.clusterresources.com/pages/products/maui-cluster-scheduler.php.

[4] PBS Professional. http://www.altair.com/software/pbspro.htm.

[5] TORQUE Resource Manager. http://www.clusterresources.com/pages/products/torque-resource-manager.php.

[6] The Grid Workloads Archive. http://gwa.ewi.tudelft.nl, 2007.

[7] S. Bansala, P. Kumarb, and K. Singh. An improved two-step algorithm for task and data parallel scheduling in distributed memory machines. *Parallel Computing*, 32(10):759–774, 2006.

[8] V. Boudet, F. Desprez, and F. Suter. One-Step Algorithm for Mixed Data and Task Parallel Scheduling Without Data Replication. In *17th Int. Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 2003.

[9] N. Capit, G. D. Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. A batch scheduler with high-level components. In *Proc. of the Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, May 2005.

[10] H. Casanova. Benefits and Drawbacks of Redundant Batch Requests. *Journal of Grid Computing*, 5(2):235–250, 2007.

[11] C. Castillo, G. N. Rouskas, and K. Harfoush. Efficient Implementation of Best-Fit Scheduling for Advance Reservations and QoS in Grid. In *Proc. of the 1st IEEE/IFIP Intl. Workshop on End-to-end Virtualization and Grid Management (EVGM)*, Oct. 2007.

[12] C. Castillo, G. N. Rouskas, and K. Harfoush. On the Design of Online Scheduling Algorithms for Advance Reservations and QoS in Grids. In *Proc. of the 21st IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, Mar. 2007.

[13] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the Benefits of Mixed Data and Task Parallelism. In *Proc. of Symp. on Parallel Algorithms and Architectures*, pages 74–83, July 1995.

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

[15] J. Decker and J. Schneider. Heuristic Scheduling of Grid Workflows Supporting Co-Allocation and Advance Reservation. In *Proc. of the Seventh IEEE Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, May 2007.

[16] L. Eyraud, G. Mounié, and D. Trystram. Analysis of Scheduling Algorithms with Reservations. In *Proc. of 21st IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS)*, Mar. 2007.

[17] U. Farooq, S. Majumdar, and E. W. Parsons. Impact of Laxity on Scheduling with Advance Reservations in Grids. In *Proc. of the the 13th IEEE Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept. 2005.

[18] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel Job Scheduling — a Status Report. In *Job Scheduling Strategies for Parallel Processing*, volume LNCS Vol. 3277. Springer-Verlag, May 2004.

[19] I. Foster and K. Chandy. Fortran M: a Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[20] M. R. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[21] Grid5000. http://www.grid5000.org, 2007.

[22] S. B. Hassen, H. Bal, and C. J. H. Jacobs. A Task- and Data-Parallel Programming Language Based on Shared Objects. *ACM Transactions on Programming Language Systems*, 20(6):1131–1170, 1998.

[23] S. Hastings, T. Kurc, S. Langella, U. Catalyurek, T. Pan, and J. Saltz. Image processing for the grid: a toolkit for building grid-enabled image processing applications. In *Proc. of the Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, pages 36–43, May 2003.

[24] C. Hu, J. Huai, and T. Wo. Flexible Resource Reservation Using Slack Time for Service Grid. In *Proc. of the 12th Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, volume 1, July 2006.

[25] S. Hunold, T. Rauber, and G. Rünger. Dynamic Scheduling of Multi-Processor Tasks on Clusters of Clusters. In *Proc. of HeteroPar'07*, Sept. 2007.

[26] K. Jansen and H. Zhang. An Approximation Algorithm for Scheduling Malleable Tasks Under General Precedence Constraints. *ACM Transactions on Algorithms*, 2(3):416–434, 2006.

[27] H. Kanazawa, M. Yamada, Y. Miyahara, Y. Hayase, S. Kawata, and H. Usami. Problem Solving Environment Based on Grid Services: NAREGI-PSE. In *Proc. of the First Intl. Conf. on e-Science and Grid Computing*, pages 456–463, Dec. 2005.

[28] C. B. Lee and A. Snavely. Precise and Realistic Utility

Functions for User-Centric Performance Analysis of Schedulers. In *Proc. of IEEE Symp. on High-Performance Distributed Computing (HPDC)*, June 2007.

[29] R. Lepere, D. Trystram, and G. Woeginger. Approximation algorithms for scheduling malleable tasks under precedence constraints. In Springer-Verlag, editor, *9th Annual European Symp. on Algorithms (ESA)*, number 2161 in LNCS, pages 146–157, 2001.

[30] M. W. Margo, K. Yoshimoto, P. Kovatch, and P. Andrews. Impact of Reservations on Production Job Scheduling. In *Proc. of the 13th Workshop of Job Scheduling Strategies for Parallel Processing*, June 2007.

[31] R. Min and M. Maheswaran. Scheduling Co-Reservations with Priorities in Grid Computing Systems. In *Proc. of the 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, May 2002.

[32] W. A. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and Runtime User Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001.

[33] H. Nakada, A. Takefusa, K. Ookubo, M. Kishimoto, T. Kudoh, Y. Tanaka, and S. Sekiguchi. Design and Implementation of a Local Scheduling System with Advance Reservation for Co-allocation on the Grid. In *Proc. of the 2006 IEEE Intl. Conf. on Computer and Information Technology (CIT)*, Sept. 2006.

[34] 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 Intl. Symp. on Parallel and Distributed Computing (ISPDC)*, July 2007.

[35] D. Nurmi, A. Mandal, J. Brevik, C. Keolbel, R. Wolski, and K. Kennedy. Evaluation of a Workflow Scheduler Using Integrated Performance Modelling and Batch Queue Wait Time Prediction. In *Proc. of SC06*, Nov. 2006.

[36] Parallel Workloads Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[37] A. Radulescu and A. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *15th Intl. Conf. on Parallel Processing (ICPP)*, Sept. 2001.

[38] A. Radulescu and A. van Gemund. A Low-Cost Approach towards Mixed Task and Data Parallel Scheduling. In *15th Int. Conf. on Parallel Processing (ICPP)*, Sept. 2001.

[39] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task- and Data-Parallelism on Distributed Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, 1997.

[40] S. Ramaswany. *Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Applications*. PhD thesis, Univ. of Illinois, Urbana-Champaign, 1996.

[41] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[42] U. Schwiegelshohn and R. Yahyapour. Fairness in parallel job scheduling. *Journal of Scheduling*, 3(5):297–320, 2000.

[43] O. Sinner. *Task Scheduling for Parallel Systems*. John Wiley, 2007.

[44] W. Smith, I. Foster, and V. Taylor. Scheduling with Advanced Reservations. In *Proc. of 14th Intl. Parallel and Distributed Processing Symp. (IPDPS)*, May 2000.

[45] Q. Snell, M. Clement, D. Jackson, and G. C. The Performance Impact of Advance Reservation Meta-Scheduling. In *Proc. of the 6th Workshop on Job Scheduling Strategies for Parallel Processing*, volume LNCS Vol. 1911. Springer-Verlag, June 2000.

[46] T. Stef-Praun, B. Clifford, I. Foster, U. Hasson, M. Hategan, S. Small, M. Wilde, and Y. Zhao. Accelerating Medical Research using the Swift Workflow System. In *Proc. of HealthGrid*, Apr. 2007.

[47] 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 *Proc. of the Intl. Conf. on Parallel Processing (ICPP)*, Aug. 2006.

[48] H. Zhao and R. Sakellariou. Advance Reservation Policies for Workflows. In *Job Scheduling Strategies for Parallel Processing*, volume LNCS Vol. 4376. Springer-Verlag, May 2006.