International Journal of Foundations of Computer Science

SCHEDULING ON LARGE SCALE DISTRIBUTED PLATFORMS: FROM MODELS TO IMPLEMENTATIONS

PIERRE-FRANÇOIS DUTOT, LIONEL EYRAUD,
GRÉGORY MOUNIÉ and DENIS TRYSTRAM*
Laboratoire ID-IMAG, Institut National Polytechnique de Grenoble,
51 avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

Today, large scale parallel systems are available at low cost. Many powerful such systems have been installed all over the world and the number of users is always increasing. The difficulty of using them efficiently is growing with the complexity of the interactions between more and more architectural constraints and the diversity of the applications. The design of efficient parallel algorithms has to be reconsidered under the influence of new parameters of such platforms (namely, cluster, grid and global computing) which are characterized by a larger number of heterogeneous processors, often organized in several hierarchical sub-systems. At each step of the evolution of the parallel processing field, researchers designed adequate computational models whose objective was to abstract the real world in order to be able to analyze the behavior of algorithms.
In this paper, we will investigate two complementary computational models that have been proposed recently: Parallel Task (PT) and Divisible Load (DL). The Parallel Task (i.e. tasks that require more than one processor for their execution) model is a promising alternative for scheduling parallel applications, especially in the case of slow communication media. The basic idea is to consider the application at a coarse level of granularity. Another way of looking at the problem (which is somehow a dual view) is the Divisible Load model where an application is considered as a collection of a large number of elementary – sequential – computing units that will be distributed among the available resources. Unlike the PT model, the DL model corresponds to a fine level of granularity. We will focus on the PT model, and discuss how to mix it with simple Divisible Load scheduling.
As the main difficulty for distributing the load among the processors (usually known as the scheduling problem) in actual systems comes from handling efficiently the communications, these two models of the problem allow us to consider them implicitly or to mask them, thus leading to more tractable problems.
We will show that in spite of the enormous complexity of the general scheduling problem on new platforms, it is still useful to study theoretical models. We will focus on the links between models and actual implementations on a regional grid with more than 500 processors.

Keywords: Scheduling, Models

*Contact Author: Denis TRYSTRAM (Denis.Trystram@imag.fr)

1. Introduction

In the Parallel Processing area, scheduling is a crucial problem for determining the starting times of the tasks and the processor locations. Many theoretical studies were conducted [5] and some efficient practical tools have been developed for old generation shared-memory systems [23, 37] or for loosely coupled distributed memory parallel machines made of homogeneous processors and sophisticated interconnection networks [20].

Scheduling in modern parallel and distributed systems is much more difficult because of new characteristics of these systems. These last few years, super-computers have been replaced by collections of large number of standard components, physically far from each other and heterogeneous [13]. This trend is clearly visible in the data provided by the Top500 organization [35], where the number of clusters and constellations started from a few percent in 1999 to reach almost three quarters of the 500 most powerful computer systems today.

The need of efficient algorithms for managing these resources is a crucial issue for a more popular use. Today, the lack of adequate software tools is the main obstacle for using these powerful systems in order to solve large and complex applications.

Our contribution in this work is to study the adequateness of models to actual implementation on large scale parallel platforms. We focus on two existing models underlying resource management systems.

In the next section, we review some classical theoretical models, to introduce the platform model we are interested in. In the following section we present two recent models that have been designed for specific applications/platforms pairs, namely Divisible Load, which considers very regular fine grain applications on complex platforms and Parallel Tasks, which considers complex applications on homogeneous platforms. In Section 4, we detail several interesting criteria for the resource management problem. Then we summarize in Section 5 some theoretical results on Parallel Tasks to put in perspective the experiments in Section 6. In Section 7, we discuss a framework for extending these results to general *light grids*. Finally, some concluding remarks are provided in Section 8.

2. Parallel Processing Today

2.1. Theoretical Models

PRAMs is the oldest abstraction of parallel machines [21]. Many extensions have been proposed to capture more realistic features like asynchronous, large grain computations, and communications. A comparative survey can be found in [2].

The classical scheduling algorithms that have been developed for parallel machines of the nineties are not well adapted to new execution platforms, as the most important factor is the influence of communications.

The first attempt that took into account the communications into computational models was to adapt and refine existing models into more realistic ones (delay model

with unitary delays [27], LogP model [12]). However, even the most elementary problems are already intractable [36], especially for large communication delays. The other characteristics of the new execution platforms are heterogeneity of processors or communication media, several levels of hierarchy (from SMP nodes to clusters and grids), versatility of the system components (some nodes can appear or disappear, new jobs can be created at any moment depending of the results of a job, etc.).

Starting from the new execution models, we look for practical tools for efficient resource management. We consider the notion of *light grid* as a collection of few clusters in a same geographical area. It is an intermediate step for a better understanding of general grids and global computing. We propose a pragmatic approach which is based on several years of experience using a 225 PC cluster at IMAG and the regional grid CiGri, which gather more than 500 machines [10].

2.2. Description of the Platform Model

The target execution platform that we consider here is a few clusters composed each by a collection of a reasonable number of SMP or simple PC machines (typically several tenths or several hundreds of nodes). Such a system may be highly heterogeneous between clusters (different kinds of processors, different numbers of processors, different Operating Systems, etc.), but weakly heterogeneous inside each cluster (different generations of processors running under the same Operating System with different clock speeds). No specific topology is assumed, but the interconnection network is fast and may be hierarchical.

Fig. 1. Synthetic view of a light grid.

The submissions of jobs is done by some specific nodes by the way of several priority queues. No other submission is allowed. Each cluster is administrated by a separate system administrator but of course, ad-hoc cooperative agreements have been established between resource owners.

2.3. How to Manage the Resources?

There is no global consensus today for an universal way of looking at the resource management problem on grids. Adequate computational models have to be

developed for designing and analyzing scheduling algorithms.

The delay models [29], based on explicit handling of communications, often cannot be used because of their intrinsic intractability. Two alternative models have been proposed, namely the Divisible Load model (DL) and the Parallel Tasks model (PT).

We believe that there is no easy way for determining a standard solution for managing the resources. As the objectives may be different from one community to another, it seems impossible to formalize the global problem as a classical combinatorial optimization problem. Thus, studies on simplified theoretical models lead to well-founded algorithms with some guaranties, which can be used in actual settings with satisfying results even if the global theoretical bounds do not hold.

3. Alternative Models

3.1. Divisible Load – DL

A Divisible Load can be seen as a (usually large) set of computations that can be partitioned in every possible way, each part being completely independent of the other parts. This model was first introduced in [8] for the processing of big data files on arrays of processors.

As each part has to be completely independent, the jobs modeled with the DL model cannot have data dependencies or communication within the tasks. With the partitioning property, the share sent to a processor may be very small with respect to the total work (fine grain). Since the introduction of this model, many kinds of applications have been considered as Divisible Loads, such as parametric executions or image and signal processing.

The DL model is more interesting than precedence constrained PT tasks for heterogeneous platforms and slow communications, as no communications occur during the computation of the task (each share is independent). The difficulty of scheduling lies in the distribution of the task to the available processors. This distribution can be made in one, several rounds[a] or dynamically with a work stealing strategy [6].

Many kinds of processor topologies have been studied such as homogeneous trees [9], heterogeneous bus [34] or stars [7] for the single round problem. Simple problems as the single round distribution on processors connected by a common bus are polynomial, but the complexity becomes quickly NP-hard with more general network topologies.

At the end of the computation, if we are for example searching something in a database there is only one processor which has to send back data. However, if all the data processing produces output, the communications gathering the results can be done as a mirror image of the data distribution. Unfortunately, mirrored schedules are usually not optimal, even if the original schedule which is mirrored is optimal.

[a] A round is a time interval in which every processor can get at most one chunk of load.

With several rounds of communications, a common approach consists in defining asymptotically optimal algorithms, where the makespan gets closer to the optimal as the number of rounds increases (as in [38]). In order to achieve this result, the authors of [4] maximized the throughput[b] to get a simple distribution scheme which is repeated at will.

3.2. Parallel Tasks – PT

Informally, a Parallel Task (PT) is a *task* that gathers elementary operations, typically a numerical routine or a nested loop, which contains itself enough parallelism to be executed by more than one processor.

We consider PT as independent jobs (applications) submitted in a multi-user context. Usually, new PT are submitted at any time (on-line). The time for each PT can be estimated or not (clairvoyant or not) depending on the type of applications. We will consider mainly the first case in this paper: we have an estimation of the characteristics of the submitted jobs (expected running times, parallel profile – at least qualitatively, etc.).

The PT model seems particularly well-adapted to grid and global computing because of the intrinsic characteristics of these new types of platforms: namely, large communication delays which have a smaller impact since the granularity is increased, moreover communications are considered implicitly and not explicitly as in all standard models leading to more robust algorithms, finally the hierarchical character of the execution platform which may be naturally expressed in the PT model as discussed in [19] (an extra constraint on the allotment of tasks is added to use neighbor processors).

We usually distinguish between three types of Parallel Tasks (PT):

- *Rigid* jobs when the number of processors to execute the PT is fixed a priori. In this case, the PT can be represented as a rectangle in a Gantt chart. The allocation problem corresponds to a strip-packing problem [28].
- *Moldable* jobs when the number of processors to execute the PT is not fixed but determined before the execution. As in the previous case this number does not change until the completion of the PT.
- *Malleable* jobs when the number of processors may change during the execution (by preemption of the tasks or simply by data redistribution).

For historical reasons, most of submitted jobs are rigid. However, intrinsically, most parallel applications are moldable. An application developer does not know in advance the exact number of processors which will be used at run time. Moreover, this number may vary with the input problem size or number of available nodes. This is also true for many numerical parallel libraries. The main restriction is the minimum number of processors that are needed because of time, memory or storage constraints.

[b] A short definition of throughput is given in Section 4.

The main restriction in a systematic use of the moldable property is the need for a practical and reliable way to estimate (at least roughly) the parallel execution time as function of the number of processors. Most of the time, the user has this knowledge but this is an inertia factor against the more systematic use of such models. Most parallel programming tools or languages have some malleability support, with dynamic addition of processing nodes support. Modern well-known advanced parallel programming environments, like Condor, Globus or Mosix implement advanced capabilities, including resilience, preemption, migration, or at least the model allows us to implement these features.

Malleability is much more easily usable from the scheduling point of view but it requires advanced capabilities from the runtime environment, and thus restricts the use of such environments and their associated programming models. In the near future, moldability and malleability should be used more and more. We will not consider malleability here, but focus on moldability as a first step.

4. Optimization Criteria

The main objective function used historically is the *makespan*. This function measures the ending time of the schedule, i.e., the latest completion time over all the tasks. However, this criterion is interesting only if we consider the tasks altogether and from the viewpoint of a single user. If the tasks have been submitted by several users, as it is the case for grid computing, other criteria have to be considered. Let us introduce some notations and review briefly various possible criteria usually used in the literature.

Notations and execution constraints

A processor executes only one task at a time. Any task i is scheduled without preemption from time $\sigma(i)$ on $nbproc(i)$ processors, with an execution time $p_i(nbproc(i))$. $nbproc$ can be a vector in the case of specific allocations for heterogeneous processors. The task i completes at time C_i equals to $\sigma(i) + p_i(nbproc(i))$. It is scheduled after its release date r_i and the scheduler may try to schedule it before its (mandatory or not) due date d_i. All values are assumed to be positive.

Criteria

- Minimization of the *makespan* ($C_{max} = max(C_i)$). This criterion is well-known as an algorithm was proposed in 1966 with constant approximation ratio [24]. The main idea of the heuristic is to realize a trade-off between the total amount of work to execute and the critical path. The constrained version of Graham's algorithm can be adapted to parallel tasks.

  Optimizing this criterion corresponds to maximizing the utilization of the computing resources in a global point of view. However it is probably insufficient in multi-user environments. Moreover, in the off-line case this is equivalent to maximizing the throughput criterion in the makespan interval.

- Minimization of the average completion time (ΣC_i) [32, 1], often written $minsum$, and its variant with weights ($\Sigma \omega_i C_i$). Giving weights to tasks al-

lows the users or system administrators to define priorities between tasks. For instance, big weights can be assigned to urgent tasks in order to reduce their completion times. For example, when a large task and many small tasks have to be scheduled concurrently, delaying the large task in order to compute the small tasks first is generally a good solution, because the large task is not delayed too much whereas the improvement on the completion time of the small ones is important.

- Minimization of the mean *stretch* (defined as the sum of the difference between completion times and release dates: $\Sigma C_i - r_i$). In an on-line context it represents the average response time between the submission and the completion. Of course, if $\forall i, r_i = 0$, for example in a batch context, it is equivalent to ΣC_i. Additionally, the value of stretch is the value of *minsum* minus the constant $\sum r_i$. Thus the optimal solution of one criterion is also the optimal solution of the other. Nevertheless, the performance ratios for both criteria for an algorithm are not as easily comparable. The performance ratio of an algorithm on stretch is also the performance ratio of this algorithm for *minsum*, but the inverse is not true.

- Minimization of the maximum stretch (i.e. the longest waiting time for a user). This criterion allows us to obtain guaranties close to "real time" systems with rejection. It does not seem to be completely relevant in grid computing.

- Maximum throughput (or steady state) defined as the maximum number of elementary tasks to execute in a given amount of time or for asymptotically long times. It is well-suited for some types of jobs like parametric computations [4]. This criterion is also useful for problems with rejection, e.g. with mandatory due date.

- Tardiness. Each task is associated with an expected due date and the schedule must minimize either the number of late tasks, the sum of the tardiness ($\sum(C_i - d_i)$ for all late tasks i) or the maximum tardiness ($\max_i(C_i - d_i)$). Minimizing the number of late tasks may produce starvation by delaying forever one particular task, thus it should be used with great care (for instance in a multi-criteria analysis, together with another criterion). The sum of tardiness and the maximum of tardiness are close to minsum and makespan but introduce in addition a partitioning of the tasks between late and not late tasks, which add an extra cost.

- Fairness. This criterion is rather a qualitative measure which is difficult to define accurately and numerous definitions have been proposed. Most of the time, the users want to share evenly computation time and/or computing resources. However sometimes the owner wants to fully control his computing resources whenever he needs, maybe even killing all jobs from non-owners. In this scenario, non-owners use the computing resources only when they are not used by the owner. Applications inspired from "Seti@home" are typical examples.

Another way of looking at the fairness is to consider several users sharing a single computing resource, without a privileged user as the owner. The fairness is then a fair division over time of the resource.

Other criteria may include rejection of tasks or normalized versions (with respect to the workload) of the previous ones.

5. Some Results about Parallel Tasks

We concentrate in this section on the PT model. We will show some interesting results that can be combined together in order to construct realistic scheduling algorithms.

In the PT model, communications are taken into account via a global *penalty* factor that reflects the overhead for data distributions, synchronization, preemption or any extra factors coming from the management of the parallel execution. The penalty factor implicitly takes into account some constraints, when they are unknown or too difficult to estimate formally. It can be determined by empirical or theoretical studies (benchmarking, profiling, performance evaluation through modeling or measuring, etc.).

We first consider the case of a single cluster. Independent jobs have been submitted to a file and are ready to be executed. More formally, we consider the off-line scheduling of a set of n independent moldable jobs on m identical processors with the objective of minimizing the makespan. Most of the existing methods for solving this problem have a common geometrical approach by transforming the problem into 2 dimensional packing problems. It is natural to decompose the problem in two successive phases: determining first the number of processors for executing the jobs, then solve the corresponding scheduling problem with rigid jobs.

5.1. A Good Off-line Approximation Algorithm

We recall briefly the principle on the best known algorithm for solving this problem [17]. The idea is to determine the job allocation with great care in order to fit them into a particular packing scheme that is inspired from the shape of the optimal one.

The MRT algorithm has a performance ratio of $3/2 + \epsilon$ [17]. It is obtained by stacking two shelves of respective sizes λ and $\frac{\lambda}{2}$ where λ is a guess of the optimal value C^*_{max}. This guess is computed by a dual approximation scheme [26]. A binary search on λ allows us to refine the guess with an arbitrary accuracy ϵ.

The guess λ is used to bound some parameters on the tasks. We give below some constraints that are useful for proving the performance ratio. In the optimal solution, assuming $C^*_{max} = \lambda$:

- $\forall j$, $p_j(nbproc(j)) \leq \lambda$.
- $\sum w_j(nbproc(j)) \leq \lambda m$.

- When two tasks share the same processor, the execution of one of these tasks is lower than $\frac{\lambda}{2}$. As there are no more than m processors, less than m processors are used by the tasks with an execution time larger than $\frac{\lambda}{2}$.

This MRT algorithm is the basis of an on-line algorithm described in the next section.

5.2. On-line Batch Scheduling

An important characteristic of the new parallel and distributed systems is the versatility of the resources: at any moment, some processors (or groups of processors) can be added or removed. At the same time, the increasing availability of the clusters or collections of clusters involved new kind of data intensive applications (like data mining) whose characteristics are that the computations depend on the data sets. The scheduling algorithm has to be able to react step by step to arrival of new tasks and thus, off-line strategies can not be used. Depending on the applications, we distinguish two types of on-line algorithms, namely, clairvoyant on-line algorithms when most parameters of the Parallel Tasks are known as soon as they arrive, and non-clairvoyant ones when only a partial knowledge of these parameters is available.

Most of the studies about on-line scheduling concern independent tasks, and more precisely the management of parallel resources. We invite the reader to look at the survey [31] for more details about on-line algorithms. In this section, we consider the clairvoyant case, where an estimate of the task execution time is known.

We first recall a generic result about *batch scheduling*. In this context, the jobs are gathered into sets (called batches) that are scheduled together. All further arriving tasks are delayed to be considered in the next batch. This is a nice way for dealing with on-line algorithms by a succession of off-line problems. We will use the result of Shmoys et al. [33] who proposed how to adapt an algorithm for scheduling independent tasks without release dates (all tasks are available at date 0) with a performance ratio of ρ into a batch scheduling algorithm with unknown release dates with a performance ratio of 2ρ.

Now, using the off-line algorithm with a performance ratio of $3/2 + \epsilon$, it is possible to schedule moldable independent tasks with release dates with a performance ratio of $3 + \epsilon$ for C_{max}. The algorithm is a batch scheduling algorithm, using the previous independent tasks algorithm at every phase.

The makespan criterion does not always have a clear meaning, especially for very long execution windows. The users usually prefer to have a guaranty that in average, their jobs are performed in the minimum time.

5.3. Batch Scheduling for Average Completion Time

Scheduling to minimize the average completion times ($minsum$) is very different

than scheduling to minimize the makespan. Good scheduling algorithms for one criterion usually have a very big performance ratio for the other criterion. The single machine problem has a polynomial optimal solution which consists of sorting the tasks by increasing sizes and scheduling them in this order. In the weighted case, where each task is associated to a weight (defining its priority), the scheduling is done according to the time/weight ratio.

In the off-line multi-processor case, scheduling with batches (or shelves) allows us to return to this simple single machine problem. Each batch has a time length and a weight (the sum of the weight of their tasks) and finding the optimal order of batches is exactly the single machine problem. Schwiegelshohn et al. [30] proposed for rigid PTs to use shelves (where all the tasks start at the same time) filled with tasks of approximately the same length (shelves sizes are powers of 2). The performance ratio is 8 for the unweighted case and 8.53 for the weighted case.

The shelves here were just filled with a well-known first fit algorithm. We will see that this ratio can be improved using more complex scheduling algorithms within batches instead of stacking tasks on shelves.

5.4. Bi-criteria Analysis

As said before, several criteria could be used to describe the quality of a schedule. The choice of which criterion to choose depends on the users view of the problem or the system administrators point of view.

However, one could wish to take advantage of several criteria in a single schedule. We present here such an analysis for the two most relevant criteria (which are somehow antagonistic). With the makespan and the sum of weighted completion times, it is easy to find examples where there is no schedule reaching the optimal value for both criteria. We can try to study how far the solution of a schedule is from the optimal one for each criterion. In this section, we present a specific family of scheduling algorithms for independent on-line moldable tasks.

There exists an approach for obtaining a bi-criteria algorithm starting from two algorithms for each criterion. It is also possible to design an ad hoc bi-criteria algorithm just by adapting an algorithm $\mathcal{A}_{C_{max}}$ designed for the makespan criterion [25]. This solution is better and is detailed below.

The main idea is to use algorithm $\mathcal{A}_{C_{max}}$ (with performance ratio $\rho_{C_{max}}$ on the makespan) as a procedure to build a schedule which has a performance guaranty on the sum of the completion times. The makespan algorithm $\mathcal{A}_{C_{max}}$ takes as input a set of (possibly weighted) tasks and a deadline d, and outputs a schedule of length at most $\rho_{C_{max}} d$ with as many tasks as possible (or the maximum weight).

Running this $\mathcal{A}_{C_{max}}$ algorithm iteratively in batches of doubling sizes (d, $2d$, $4d$, ...) gives a schedule where the total makespan is at most $4\rho_{C_{max}} C^*_{max}$ as the last batch is smaller than $2\rho_{C_{max}} C^*_{max}$. The performance ratio on the sum of completion times is also $4\rho_{C_{max}}$. The technical proofs are in the original article [25].

The resulting schedule is depicted in Figure 2. The tasks are chosen at each step among the available tasks, and scheduled in batch of doubling sizes.

Fig. 2. Principle of the bi-criteria algorithm using batches of doubling sizes.

6. Simulation of a Practical Use

Due to several technical constraints, the implementation of the previous algorithm in an actual cluster environment requires to modify it. While we maintained the overall structure in batches of doubling sizes, the algorithm used to pack each batch was changed so as to fit in an environment using the classic FIFO with backfilling strategy. With this change, (almost)[c] all the tasks in one batch start their execution at the beginning of the batch – which means tasks are not "piled up" on top of each other, and thus the sum of the allocations of the tasks do not exceed the number of processors. With such an allocation, the packing of tasks in a batch is much simpler, and therefore easier to implement in an actual environment. The downside of this approach is that we have not been able to keep a performance guarantee for this modified algorithm, although simulation results show that it has a good average performance.

6.1. Experimental setting

The experimental simulations presented here were performed with an ad-hoc program. Each experience is repeated 40 times; for each run tasks are generated in an off-line manner, then given as an input to the scheduling algorithm and to the linear program solver which computes a lower bound for this instance. Comparison between the two results yields a performance ratio, and the average ratio for the whole set of runs is the result of the experiments.

The runs were made assuming a cluster of 200 processors, and a number of tasks varying from 25 to 200. In order to describe a mono-processor task, only its computing time is needed. A moldable task is described by a vector of m processing times (one per number of processors alloted to the task). We used two different classical models to generate the tasks. The first one generates the sequential processing times of the tasks, and the second one uses a parallelism

[c]The exception concerns sequential tasks whose execution times are smaller than half of the batch length of the batch, which can be placed one after the others.

model to derive all the other values.

Two different sequential workload types were used: uniform and mixed cases. For all uniform cases, sequential times were generated according to an uniform distribution, varying from 1 to 10. For mixed cases, we introduce two classes: small and large tasks. The random values are taken with Gaussian distributions centered respectively on 1 and 100, with respective standard deviations of 0.4 and 40, the ratio of small tasks being 70%.

Modeling the parallelism of the jobs was done in two different ways. In the first one, successive processing times were computed with the formula $p_i(j) = p_i(j-1)\frac{X+j}{1+j}$, where X is a random variable between 0 and 1. Depending on the distribution of X, tasks generated are highly parallel (with a quasi-linear speedup) or weakly parallel (with a speedup close to 1). Respectively highly and weakly parallel are generated using gaussian distribution centered on 0.9 and 0.1, and with a standard deviation of 0.2. Any random value smaller than 0 and larger than 1 is ignored and recomputed. According to the usual parallel program behavior, this method generates monotonic tasks, which have decreasing execution times and increasing work with the number of alloted processors. For the mixed cases, the small tasks are weakly parallel and the large tasks are highly parallel.

The second way of modeling parallelism was done according to a model from Cirne and Berman [11], which relies on an analysis of the behavior of the users in a computing center.

To evaluate our algorithm, we use a lower bound obtained by a relaxed linear program as reference. Some simple "standard" list algorithms are used to compare the behavior and efficiency of our approach. All the 3 algorithms use multiprocessor list scheduling [22]. Every task is alloted using the number of processors selected by [16]. This should lead to a very good average performance ratio with respect to the C_{max} criterion. Only the order of the list is changed between the three algorithms:

- the first one keeps the order of [16], listing first the tasks of the large shelf, then the tasks of the small shelf, then the small tasks.
- weighted largest processing time first (LPTF), a classical variant, with a very good behavior for C_{max} criterion; but the tasks are actually sorted using the ratio between weighted and their execution time.
- smallest area first (SAF), almost the opposite of LPTF, the tasks are sorted according to their area (number of processors $\times$ execution time). The goal is to improve the average performance ratio for the $\sum w_i C_i$ criterion.

In all experiments, the task priority is a random value taken from an uniform distribution between 1 and 10.

6.2. Comparing to other algorithms

Figure 3 shows the result of the experiments for both criteria using the Cirne workload model, which tries to emulate real applications. On this figure, our bicriteria algorithm is named DEMT. We can see that in this setting, our algorithm

Fig. 3. Performance ratio for the simulation on 200 processors, parallel tasks following the Cirne model

significantly outperforms the other algorithms for the minsum criterion, especially for a low number of tasks. This is not true for C_{max}, but the difference is much less important.

When using other workload models, we can make the following observations (see [15] for more details):

- List algorithms perform always better than our algorithm as far as C_{max} is concerned. This was expected, since our main focus was the $\sum \omega_i C_i$ criterion. However, the performance ratio for the C_{max} criterion is always less than 2.
- A workload made only of weakly parallel tasks is the worst case for our algorithm, which can be explained by the fact that the algorithm spends a lot of the resources trying to accelerate the completion of small and high priority tasks. Since they are not very parallel, the gain is too small and some resources are wasted. However, the performance ratio of our algorithm is always less than 2 for C_{max}, and less than 2.5 for $\sum \omega_i C_i$, even in this worst case scenario.
- On more parallel workloads, our algorithm behaves much better for the $\sum \omega_i C_i$ criterion. It has the best ratio of all algorithms for the higly parallel workloads, and is only dominated by SAF on the mixed workload.

6.3. Varying Parameters

A natural question that arises when implementing this algorithm is wether doubling the size of the batch at each iteration is a "good" choice. The main idea in this algorithm is to have batches of exponentially increasing sizes, but the actual value α by which we should multiply the size of the batch at each iteration is not fixed. This parameter actually represents a tradeoff between the two criteria C_{max} and $\sum \omega_i C_i$. With small values of α, batches stay relatively small at the beginning, and the algorithm focuses more on scheduling small and high-weight tasks, thus improving the $\sum \omega_i C_i$ criterion. On the other hand, large values of α imply larger batches, so the algorithm focuses more on scheduling efficiently a high number of tasks – improving on the C_{max} criterion.

A theoretical study was made on this topic (see [18]). As far as worst cases are concerned, the best value for the $\sum \omega_i C_i$ criterion is to have $\alpha = 2$, with larger values improving C_{max} but degrading $\sum \omega_i C_i$, and smaller values degrading both criteria. However, when considering average behavior rather than worst case, we observe a similar behaviour, with the limit value being $e \simeq 2.718$.

To evaluate the influence of this parameter on our implementation of the algorithm, we conducted a series of experiments in the same setting as described earlier. The parameter α was set to vary between 1.05 and 6.3. Figures 4 and 5 show the results of this simulation, in the uniform and mixed workloads respectively, both using the Cirne parallelism model. On each figure, the right-hand part shows the performance ratios for both criteria as a function of α, and the left-hand part is

Fig. 4. Influence of the α parameter, with uniform workload

Fig. 5. Influence of the α parameter, with mixed workload

a plot of the ratio for C_{max} against the ratio for $\sum \omega_i C_i$, when α varies. This left-hand part shows the best trade-off between C_{max} and $\sum \omega_i C_i$.

We have selected two extreme cases for the number of tasks in the system, namely 25 and 200. With 25 tasks, there is often one which is not parallel enough to have its processing time reduced by a significant amount, and the length of the schedule is dominated by this task. With 200 tasks, we have the opposite case where the schedule is more dominated by the total work to do.

There are several observations to make on these figures. Firstly, the optimum value of α for the $\sum \omega_i C_i$ criterion is actually lower than 2 (it is somewhere around 1.4 and 1.6). This difference between the theoretical value may be explained with the fact that the simulation results are averages whereas the theoretical studies are more interested in worst case scenarios which do not occur in our experiments. The second observation is that, unlike in the theoretical model, large values of α degrade the C_{max} performance ratio of the algorithm, especially in the uniform workload. This might again be explained by the fact that generated tasks are not different enough one from another for larger batches to be beneficial.

When using different parallel behavior, the results obtained are slightly different, and some time chaotic. For example, Figure 6 shows the result of the experiments with a mixed workload, in which there are two types of tasks: large ones and small ones, both types having a highly parallel behavior. We can see that the performance ration for C_{max} has a (reproducible) erratic behavior when α is small, between 1

Fig. 6. Influence of the α parameter, with mixed workload and high parallelism

and 2.

7. Integration into an Actual Environment

In this section, we discuss several directions for integrating the previous ideas in order to build an operational light grid management system.

7.1. Single Cluster Issues

Rigid Jobs The scheduling algorithms presented in Section 5 are targeted for *moldable* jobs. Even though most jobs are intrinsically moldable, some of them need to stay rigid, or at least cannot accept every allocation. The reason can be a lack of time to re-code the program to make it moldable, memory constraints which set a minimum number of processors, or the job can be a benchmarking job that requires a preset number of processors. This means we actually have to deal with a mix of moldable and rigid jobs.

There are different possible ideas to solve this problem. The first trivial idea is to separate rigid and moldable jobs and schedule one category after the other. Another solution is to calculate a-priori an allocation for the moldable jobs, and then apply a rigid scheduling algorithm on the resulting rigid jobs. The last solution is to modify the bi-criteria algorithm in order to schedule each rigid job in the first batch in which it fits.

Reservations An important point for a management system is the ability to perform reservations. This would allow a user to ask for a given number of processors in a given time window. Such a possibility is necessary for demonstration purposes, or in order to set up a wide-area experiment with other computing centers. The scheduling algorithm must then cope with this additional constraint, which makes a certain number of nodes unavailable during a period of time.

Including support for such reservations into a scheduling algorithm is a difficult a problem. A batch algorithm could try to ensure that batch boundaries match the beginning and the end of the reservations, but that would likely be inefficient.

7.2. Dealing with Several Clusters

This section focuses on the additional problems raised when trying to have several clusters operate together. We will first present the light grid context in which we are interested.

Fig. 7. 4 largest clusters of the CIMENT project.

The CIMENT project The system we are working on is a part of the CIMENT [10] project, in which the academic computing resources of Grenoble are connected. This results in the realization of a light grid, containing quite heterogeneous machines, more than 500 in total (see Figure 7). The goal of the project is to make different research communities (Numerical Physicists, Astrophysicists, Medical Image Researchers, Computer Scientists, ...) share their computing resources, leading to an overall better use of these resources.

Gathering different communities raises several issues. First, every community has its own behavior and culture, either for historical reasons or for reasons linked to the type of research the community performs. For example, the numerical physicists have long (up to several weeks), sequential jobs to perform, while the computer scientists' jobs are shorter, focusing mainly on debugging. Scheduling jobs with such a disparity is an issue in itself.

Furthermore, these disparities imply differences in scheduling choices. It is important to point out that each community has habits about scheduling policy, management system, submission rules, and so on. The light grid management system should try not to disturb these habits by a too large extent.

Another important point is to guarantee some notion of fairness between the different communities. Each computing resource was bought by its respective community because they wanted to use that computing power, so we should make sure that making it available to others does not make them lose too much.

A majority of the jobs submitted in this context are *multi-parametric* jobs. Such a job consists of a large number (up to several hundreds of thousands) of runs of the

same program, each having with different parameters. Each run takes a relatively short time to complete, this time being often the same for every run. This kind of jobs are related to the divisible tasks model (see Section 3.1). For this kind of jobs, the theory of asymptotic behavior shows that optimal solutions can be computed in polynomial time. This allows the use of these jobs in order to fill the holes in the Gantt chart (using the same idea as conservative backfilling).

There are different ways to link several clusters together. The first one is the current system in use in Grenoble, and the second version is rather an attempt to address the problem more globally.

Centralized In the first vision of this problem, each cluster keeps its own submission system used only for jobs that are to be processed locally. Additionally, there is a centralized server to which all grid jobs are submitted. In this setting, grid jobs are only multi-parametric jobs, which the centralized server submits on the local clusters in order to fill the holes of their respective schedules. This is achieved through the notion of *best-effort* jobs: the local scheduler gives no warranty that the job will be finished. If a locally submitted job requires a processor currently in use by a best-effort job, the latter will be killed. The central server then has to submit it once again. Since there are a large number of relatively small runs, the cost of killing one of them is not too high. Furthermore, this ensures that local users of the clusters will not be disturbed by grid jobs: they have the same submission interface as before and cannot have their job delayed by a grid job.

In order to schedule the best-effort jobs efficiently, we can rely on the Divisible Load model introduced in Section 3.1. The configuration of all the idle processors of the grid is fixed between local events such as the completion or the beginning of a local job. Therefore, if the schedules are on-line batch schedules, we can define time windows where the idle processors do not change. Within these time windows, we can pack as many parameterized computations as possible using a *makespan* algorithm. The problem of scheduling many identical tasks on complex heterogeneous topologies is NP-hard [14], however in this centralized setting, the submission node and the clusters are respectively a master node and its slaves connected as a heterogeneous fork-graph. This problem of scheduling on heterogeneous fork graph minimizing the *makespan* can be solved in polynomial time [3].

Decentralized In this vision, all jobs – grid and local ones – are submitted to local scheduling systems. These systems then have the possibility to exchange work in order to balance the load. The protocol for exchanging work still has to be defined, but it would have to take care of both fairness and performance issues at the same time. There are several directions to address this problem: namely, graph coupling which would aim at minimizing data transfers, an economical approach where each cluster can optimize its own criteria, or use consensus-driven algorithms, etc..

We do not believe that it is possible to formalize the resource management problem on grids as a global combinatorial optimization problem because of the

complexity. However we can use sophisticated algorithms for local optimization on each cluster, and then focus on alternative methods at the interface of local methods.

8. Concluding Remarks

We presented in this paper two theoretical models which concern very specific (and complementary) applications/platforms pairs, namely the Divisible Load with regular and simple computations, and the Parallel Tasks model with a high abstraction level of the machine. Using both models we described how to adapt them to actual implementations, we ran many experiments to tune parameters and we presented some of the most meaningful results.

Today there is no general scheduler for managing the resources on grids. We can expect in the next few years that intermediate level models will arise and allow to develop practical efficient schedulers.

Acknowledgements

This work was supported by the CiGri project under the national program ACI Grid2.

References

1. F. Afrati, E. Bampis, A. V. Fishkin, K. Jansen, and C. Kenyon. Scheduling to minimize the average completion time of dedicated tasks. *Lecture Notes in Computer Science*, vol. 1974, 2000.
2. E. Bampis, F. Guinand, and D. Trystram. Some models for scheduling parallel programs with communication delays. *Discrete Applied Mathematics*, 72:5–24, 1997.
3. O. Beaumont, A. Legrand, and Y. Robert. A polynomial-time algorithm for allocating independent tasks on heterogeneous fork-graphs. In *ISCIS XVII, Seventeenth International Symposium On Computer and Information Sciences*, pages 115–119. CRC Press, 2002.
4. O. Beaumont, A. Legrand, and Y. Robert. Optimal algorithms for scheduling divisible workloads on heterogeneous systems. In *Heterogeneous Computing Workshop*. IEEE computer society, 2003.
5. J. Błażewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1996.
6. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In Shafi Goldwasser, editor, *Proceedings: 35th Annual Symposium on Foundations of Computer Science, November 20–22, 1994, Santa Fe, New Mexico*, pages 356–368, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.
7. S. Charcranoon, T.G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on computers*, 49(9):987–991, September 2000.
8. Y.C. Cheng and T.G. Robertazzi. Distributed computation for a tree network with communication delays. *IEEE Trans. on Aerospace and Electronic Systems*, 24(6):700–712, 1988.

9. Y.C. Cheng and T.G. Robertazzi. *Distributed computation for a tree network with communication delays*, volume 26, pages 511–516. 1990.
10. Ciment project. http://ciment.ujf-grenoble.fr.
11. W. Cirne and F. Berman. A model for moldable supercomputer jobs. In *15th Intl. Parallel & Distributed Processing Symp.*, 2001.
12. D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.
13. D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.
14. P.-F. Dutot. Complexity of master-slave tasking on heterogeneous trees. *European Journal on Operationnal Research*, 2004. To appear.
15. P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Symposium on Parallel Algorithm and Architectures*, pages 125–132, Barcelona, 2004.
16. P.-F. Dutot, G. Mounié, and D. Trystram. *Handbook of Scheduling*, chapter Scheduling Parallel Tasks - Approximation Algorithms. Number 26. CRC press, 2004.
17. P.-F. Dutot, G. Mounié, and D. Trystram. *Handbook of combinatorics*, chapter 26. CRC Press, to appear.
18. P.-F. Dutot and D. Trystram. A best-compromise bicriteria scheduling algorithm for malleable tasks. Research report (submitted to a journal).
19. P.-F. Dutot and D. Trystram. Scheduling on hierarchical clusters using malleable tasks. In *Proceedings of the 13th annual ACM symposium on Parallel Algorithms and Architectures - SPAA 2001*, pages 199–208, Crete Island, July 2001. SIGACT/SIGARCH and EATCS, ACM Press.
20. D. G. Feitelson and L. Rudolph. Parallel job scheduling: Issues and approaches. *Lecture Notes in Computer Science*, 0(949):1–18, 1995.
21. S. Fortune and J. Wyllie. Parallelism in random access machiness. In *Proceedings of the 10th ACM Symposium on the Theory of Computing*, pages 114–118, May 1978.
22. M. R. Garey and R. L. Graham. Bounds on multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.
23. A. Gerasoulis and T. Yang. PYRROS: static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 6th ACM International Conference on Supercomputing*, pages 428–437. ACM, July 1992.
24. R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45:1563–1581, 1966.
25. L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.
26. D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.
27. J.J. Hwang, Y.C. Chow, F.D. Anger, and C.Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, April 1989.
28. A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey.

European Journal of Operational Research, 141(2):241–252, 2002.

29. V.J. Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.
30. U. Schwiegelshohn, W. Ludwig, J. Wolf, J. Turek, and P. Yu. Smart SMART bounds for weighted response time scheduling. *SIAM Journal on Computing*, 28, 1998.
31. J. Sgall. Chapter 9: On-line scheduling. *Lecture Notes in Computer Science*, 1442:196–231, 1998.
32. H. Shachnai and J. Turek. Multiresource malleable task scheduling to minimize response time. *Information Processing Letters*, 70:47–52, 1999.
33. D.B. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machine on-line. *SIAM Journal on Computing*, 24(6):1313–1331, 1995.
34. J. Sohn, T.G. Robertazzi, and S. Luryi. Optimizing computing costs using divisible load analysis. *IEEE Transactions on parallel and distributed systems*, 9(3):225–234, March 1998.
35. The top500 organization website. `http://www.top500.org`.
36. D. Trystram and W. Zimmermann. On multi-broadcast and scheduling receive-graphs under logp with long messages. In S. Jaehnichen and X. Zhou, editors, *The Fourth International Workshop on Advanced Parallel Processing Technologies - APPT 01*, pages 37–48, Ilmenau, Germany, September 2001.
37. M.-Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, 1990.
38. Y. Yang and H. Casanova. Umr: A multi-round algorithm for scheduling divisible workloads. In *International Parallel and Distributed Processing Symposium*, 2003.