Offline and online master-worker scheduling of concurrent bags-of-tasks on heterogeneous platforms

Anne Benoit [2,4,5] Loris Marchal [1,4,5] Jean-François Pineau [2,4,5]
Yves Robert [2,4,5] Frédéric Vivien [3,4,5]

[1] CNRS [2] ENS Lyon [3] INRIA [4] Université de Lyon

[5] LIP laboratory, UMR 5668, ENS Lyon – CNRS – INRIA – UCBL, Lyon, France

{anne.benoit, loris.marchal, jean-francois.pineau, yves.robert, frederic.vivien}@ens-lyon.fr

Abstract

Scheduling problems are already difficult on traditional parallel machines. They become extremely challenging on heterogeneous clusters, even when embarrassingly parallel applications are considered. In this paper we deal with the problem of scheduling multiple applications, made of collections of independent and identical tasks, on a heterogeneous master-worker platform. The applications are submitted online, which means that there is no a priori (static) knowledge of the workload distribution at the beginning of the execution. The objective is to minimize the maximum stretch, i.e. the maximum ratio between the actual time an application has spent in the system and the time this application would have spent if executed alone.

On the theoretical side, we design an optimal algorithm for the offline version of the problem (when all release dates and application characteristics are known beforehand). We also introduce several heuristics for the general case of online applications.

On the practical side, we have conducted extensive simulations and MPI experiments, showing that we are able to deal with very large problem instances in a few seconds. Also, the solution that we compute totally outperforms classical heuristics from the literature, thereby fully assessing the usefulness of our approach.

1. Introduction

Scheduling problems are already difficult on traditional parallel machines. They become extremely challenging on heterogeneous clusters, even when embarrassingly parallel applications are considered. For instance, consider a bag-of-tasks [1], i.e., an application made of a collection of independent and identical tasks, to be scheduled on a master-worker platform. Although simple, this kind of framework is typical of a large class of problems, including parameter sweep applications [6] and BOINC-like computations [5]. These middlewares usually organize participating processors in a master/worker platform, and make use of dynamic schedulers. When the platform gathers heterogeneous processors, connected to the master via different-speed links, then purely demand-driven approaches are likely to fail dramatically. This is because it is crucial to select which resources to enroll before initiating the computation [2].

In this paper, we still target fully parallel applications, but we introduce a much more complex (and more realistic) framework than scheduling a single application. We envision a situation where users, or clients, submit several bags-of-tasks to a heterogeneous master-worker platform, using a classical client-server model. Applications are submitted online, which means that there is no a priori (static) knowledge of the workload distribution at the very beginning of the execution.

When several applications are executed simultaneously, they compete for hardware (network and CPU) resources. What is the scheduling objective in such a framework? A greedy approach would execute the applications sequentially in the order of their arrival, thereby optimizing the execution of each application onto the target platform. Such a simple approach is not likely to be satisfactory for the clients. Sharing resources to execute several applications concurrently may have two key advantages: (i) from the clients' point of view, the average response time is expected to be much smaller; (ii) from the resource usage perspective, the global utilization of the platform is likely to increase. The traditional measure to quantify the benefits of concurrent scheduling on shared resources is the maximum stretch. The stretch of an application is defined as the

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

ratio of its response time under the concurrent scheduling policy over its response time in dedicated mode, i.e., when it is the only application executed on the platform. The objective is then to minimize the maximum stretch of any application, thereby enforcing a fair trade-off between all applications.

The aim of this paper is to provide a scheduling strategy which minimizes the maximum stretch of several concurrent bags-of-tasks which are submitted online. Our scheduling algorithm relies on complicated mathematical tools but can be computed in time polynomial to the problem size. On the theoretical side, we prove that our strategy is optimal for the offline version of the problem (when all release dates and application characteristics are known beforehand). We also introduce several heuristics for the general case of online applications. On the practical side, we have conducted extensive simulations and MPI experiments, showing that we are able to deal with very large problem instances in a few seconds and still outperform classical heuristics from the literature, thereby fully assessing the usefulness of our approach.

The rest of the paper is organized as follows. Section 2 describes the platform and application models. Section 3 is devoted to the derivation of the optimal solution in the offline case, and to the presentation of heuristics for online applications. In Section 4 we report an extensive set of simulations and MPI experiments, and we compare the optimal solution with several classical heuristics from the literature. Finally, we state some concluding remarks in Section 5. Due to lack of space, the review of related work is provided in the dedicated research report [4].

2. Framework

In this section, we outline the model for the target platforms, as well as the characteristics of the applicative framework. Next we survey steady-state scheduling techniques and we introduce the objective function.

2.1. Platform Model

We target a heterogeneous master-worker platform (see Figure 1), also called star network or single-level tree in the literature.

The master P_{master} is located at the root of the tree, and there are p workers P_u ($1 \leq u \leq p$). The link between P_{master} and P_u has a bandwidth b_u. We assume a linear cost model, hence it takes X/b_u time-units to send (resp. receive) a message of size X bits to (resp. from) P_u. The computational speed of worker P_u is s_u, meaning that it takes X/s_u time-units to execute X floating point operations. Without any loss of generality, we assume that the master has no processing capability.

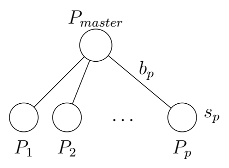

Figure 1. A star network.

The communication model is the *bounded multi-port* model [10]: the master can send/receive data to/from all workers at a given time-step. However, there is a limit on the amount of data that the master can send per time-unit, denoted as BW. Simultaneous sends and receives are allowed (all links are assumed bi-directional, or full-duplex). Finally, we assume that computation can be overlapped by independent communication, without any interference.

For computations, we enforce the *fluid computation* model, where several tasks can be executed concurrently on a given worker. Furthermore, we assume that we totally control the computation rate for each task, for example using a time-sharing mechanism. These computation rates may be changed at any time during the computation of a task.

Unlike classical master/workers models, we do not assume that a worker has to wait for a task to be completely received before starting its computation. On the contrary, we assume that the computation of a task can start at the same time as its transfer, modeling the fact that, in several applications, only the first bytes of data are needed to start its execution [11]. This model is much more tractable than the classical ones, it provides an upper bound on the achievable performance for any model, and it is not too far from reality when considering small-size tasks[1].

2.2. Application model

We consider n bags-of-tasks A_k, $1 \leq k \leq n$. The master P_{master} holds the input data of each application A_k upon its release time. Application A_k is composed of a set of $\Pi^{(k)}$ independent, same-size tasks. In order to completely execute an application, all its constitutive tasks must be computed (in any order).

[1]For a comparison of several computation models, we refer to the extended version of the paper [4].

We let $w^{(k)}$ be the amount of computations (expressed in flops) required to process a task of A_k. As the platform can be composed of heterogeneous machines, the speed of a worker P_u may be different for each application. To take this into account, we refine the platform model and denote by $s_u^{(k)}$ the speed of worker P_u when processing application A_k. The time required to process one task of A_k on processor P_u is now $w^{(k)}/s_u^{(k)}$. Each task of A_k has a size $\delta^{(k)}$ (expressed in bytes), which means that it takes a time $\delta^{(k)}/b_u$ to send a task of A_k to processor P_u (when there are no other ongoing transfers). For simplicity we do not consider any return message.

2.3. Steady-state scheduling

Assume for a while that a unique bag-of-tasks A_k is executed on the platform. If $\Pi^{(k)}$, the number of independent tasks composing the application, is large (otherwise, why would we deploy A_k on a parallel platform?), we can relax the problem of minimizing the total execution time. Instead, we aim at maximizing the throughput, i.e., the average (fractional) number of tasks executed per time-unit. We design a cyclic schedule, that reproduces the same schedule every period, except possibly for the very first (initialization) and last (clean-up) periods. It is shown in [2] how to derive an optimal schedule for throughput maximization. The idea is to characterize the optimal throughput as the solution of a linear program over rational numbers, which is a problem with polynomial time complexity.

Throughout the paper, we denote by $\rho_u^{(k)}$ the throughput of worker P_u for application A_k, i.e., the average number of tasks of A_k that P_u executes each time-unit. In the special case where application A_k is executed alone in the platform, we denote by $\rho_u^{*(k)}$ the value of this throughput in the solution which maximizes the total throughput: $\rho^{*(k)} = \sum_{u=1}^{p} \rho_u^{*(k)}$. We write the following linear program (see Equation 1) to compute an asymptotically optimal schedule. The maximization of the throughput is bounded by three types of constraints:
• The first set of constraints states that the processing capacity of P_u is not exceeded.
• The second set of constraints states that the bandwidth of the link from P_{master} to P_u is not exceeded.
• The last constraint states that the total outgoing capacity of the master is not exceeded.

$$(1)\begin{cases} \textsc{Maximize } \rho^{*(k)} = \sum_{u=1}^{p} \rho_u^{*(k)} \text{ SUBJECT TO} \\ \forall u,\ \rho_u^{*(k)} \frac{w^{(k)}}{s_u^{(k)}} \leq 1 \\ \forall u,\ \rho_u^{*(k)} \frac{\delta^{(k)}}{b_u} \leq 1 \\ \displaystyle\sum_{u=1}^{p} \rho_u^{*(k)} \frac{\delta^{(k)}}{\text{BW}} \leq 1 \end{cases}$$

The formulation in terms of a linear program is simple when considering a single application. A closed-form expression can be derived:

$$\rho^{*(k)} = \min\left\{\frac{\text{BW}}{\delta^{(k)}}, \sum_{u=1}^{p} \min\left\{\frac{s_u^{(k)}}{w^{(k)}}, \frac{b_u}{\delta^{(k)}}\right\}\right\}.$$

It can be shown [2] that any feasible schedule has to enforce the previous constraints. Hence the optimal value $\rho^{*(k)}$ is an upper bound of the achievable throughput. Moreover, we can construct an actual schedule, based on an optimal solution of the linear program and which approaches the optimal throughput:
• While there are tasks to process on the master, send tasks to processor P_u with rate $\rho_u^{*(k)}$.
• As soon as processor P_u starts receiving a task it processes at the rate $\rho_u^{*(k)}$.

Due to the constraints of the linear program, this schedule is always feasible and it is asymptotically optimal, not only among periodic schedules, but more generally among any possible schedules. More precisely, its execution time differs from the minimum execution time by a constant factor, independent of the total number of tasks $\Pi^{(k)}$ to process [2]. This allows us to accurately approximate the total execution time, also called makespan, as:

$$MS^{*(k)} = \frac{\Pi^{(k)}}{\rho^{*(k)}}.$$

We often use $MS^{*(k)}$ as a comparison basis to approximate the makespan of an application when it is alone on the computing platform. If $MS_{\text{opt}}^{(k)}$ is the optimal makespan for this single application, then we have $MS_{\text{opt}}^{(k)} - M_k \leq MS^{*(k)} \leq MS_{\text{opt}}^{(k)}$ where M_k is a fixed constant, independent of $\Pi^{(k)}$ [2].

2.4. Stretch

We come back to the original scenario, where several applications are executed concurrently. Because they compete for resources, their throughput will be lower. Equivalently, their execution rate will be slowed down.

Informally, the stretch of an application is the slowdown factor.

Let $r^{(k)}$ be the release date of application A_k on the platform. Its execution will terminate at time $\mathcal{C}^{(k)} \equiv r^{(k)} + MS^{(k)}$, where $MS^{(k)}$ is the time to execute all $\Pi^{(k)}$ tasks of A_k. Because there might be other applications running concurrently to A_k during part or whole of its execution, we expect that $MS^{(k)} \geq MS^{*(k)}$. We define the average throughput $\rho^{(k)}$ achieved by A_k during its (concurrent) execution using the same equation as before.

In order to process all applications fairly, we would like to ensure that their actual (concurrent) execution is as close as possible to their execution in dedicated mode. The stretch of application A_k is its slowdown factor

$$\mathcal{S}_k = \frac{MS^{(k)}}{MS^{*(k)}} = \frac{\rho^{*(k)}}{\rho^{(k)}}$$

Our objective function is defined as the *max-stretch* $\mathcal{S}$, which is the maximum of the stretches of all applications: $\mathcal{S} = \max_{1 \leq k \leq n} \mathcal{S}_k$. Minimizing the *max-stretch* $\mathcal{S}$ ensures that the slowdown factor is kept as low as possible for each application, and that none of them is unduly favored by the scheduler.

3. Theoretical study

We start this section with the presentation of an asymptotically optimal algorithm for the offline setting, when application release dates and characteristics are known in advance. Then we present our solution for the online framework.

3.1. Offline setting

In this section, we assume that all characteristics of the n applications $A_k, 1 \leq k \leq n$ are known in advance.

The scheduling algorithm is the following. Given a candidate value for the max-stretch, we have a procedure to determine whether there exists a solution that can achieve this value. The optimal value will then be found using a binary search on possible values.

Consider a candidate value $\mathcal{S}^l$ for the max-stretch. If this objective is feasible, all applications will have a max-stretch smaller than $\mathcal{S}^l$, hence:

$$\forall\, 1 \leq k \leq n,\ \frac{MS^{(k)}}{MS^{*(k)}} \leq \mathcal{S}^l$$
$$\iff \forall k,\ 1 \leq k \leq n,$$
$$\mathcal{C}^{(k)} = r^{(k)} + MS^{(k)} \leq r^{(k)} + \mathcal{S}^l \times MS^{*(k)}$$

Thus, given a candidate value $\mathcal{S}^l$, we have a deadline:

$$d^{(k)} = r^{(k)} + \mathcal{S}^l \times MS^{*(k)} \qquad (2)$$

for each application A_k, $1 \leq k \leq n$. This means that the application must complete before this deadline in order to ensure the expected max-stretch. If this is not possible, no solution is found, and a larger max-stretch should be tried by the binary search.

Once a candidate stretch value $\mathcal{S}$ has been chosen, we divide the total execution time into time-intervals whose bounds are epochal times, that is, applications' release dates or deadlines. Epochal times are denoted $t_j \in \{r^{(1)}, ..., r^{(n)}\} \cup \{d^{(1)}, ..., d^{(n)}\}$, such that $t_j \leq t_{j+1}$, $1 \leq j \leq 2n - 1$. Our algorithm consists in running each application A_k during its whole execution window $[r^{(k)}, d^{(k)}]$, but with a different throughput on each time-interval $[t_j, t_{j+1}]$ such that $r^{(k)} \leq t_j$ and $t_{j+1} \leq d^{(k)}$.

Note that contrarily to the steady-state operation with only one application, in the different time-intervals, the communication throughput may differ from the computation throughput: when the communication rate is larger than the computation rate, extra tasks are stored in a buffer. On the contrary, when the computation rate is larger, tasks are extracted from the buffer and processed. We introduce new notations to take both rates, as well as buffer sizes, into account:

- $\rho_{M \to u}^{(k)}(t_j, t_{j+1})$ denotes the average number of tasks of A_k sent from the master to the worker P_u per time-unit, during time-interval $[t_j, t_{j+1}]$ (communication throughput);
- $\rho_u^{(k)}(t_j, t_{j+1})$ denotes the average number of tasks of A_k computed by P_u per time-units, during time-interval $[t_j, t_{j+1}]$ (computation throughput);
- $B_u^{(k)}(t_j)$ denotes the (fractional) number of tasks of application A_k stored in a buffer on P_u at time t_j.

We write the linear constraints that must be satisfied by the previous variables. Our aim is to find a schedule with minimum stretch satisfying those constraints.

The additional constraints are summarized here:

- all the tasks of a given application A_k are sent by the master,
- each buffer should always have a non-negative size,
- at the beginning of the computation of application A_k, all corresponding buffers are empty,
- after the deadline of application A_k, no tasks of this application should remain on any node,
- during time-interval $[t_j, t_{j+1}]$, some tasks of application A_k are received and some are consumed (computed), which affects the size of the buffer,
- all throughputs are non-negative.

We still have to add the original constraints from Linear Program (1) to ensure the platform capacity is not exceeded. Please refer to the companion research report [4] for a detailed presentation of these constraints. We obtain a convex polyhedron (K) defined by the previous constraints. The problem turns now into checking whether the polyhedron is empty and, if not, into finding a point in the polyhedron.

$$\begin{cases} \rho_{M\to u}^{(k)}(t_j, t_{j+1}), \rho_u^{(k)}(t_j, t_{j+1}), \\ \forall k, u, j \text{ such that } 1 \leq k \leq n, 1 \leq u \leq p \\ \text{under all previous constraints} \end{cases} \quad \text{(K)}$$

One can note that we can easily add some constraints to bound the buffers to any prescribed maximal size.

Theorem 1. *Polyhedron* (K) *is not empty if and only if there exists a schedule with stretch $\mathcal{S}$.*

The proof of this theorem can be found in [4]. In practice, to know if the polyhedron is empty or to obtain a point in (K), we can use classical tools for linear programs. Finding a point in Polyhedron (K) allows to determine whether the candidate value for the stretch is feasible. Depending on whether Polyhedron (K) is empty, the binary search will be continued with a larger or smaller stretch value.

The initial upper bound for this binary search is computed using a naive schedule where all applications are computed sequentially. For sake of simplicity, we consider that all applications are released at time 0 and terminate simultaneously. This is clearly a worst case scenario, and allows us to compute the maximum stretch $\mathcal{S}_{\text{max}}$. The lower bound on the achievable stretch is 1.

We suppose here the termination criterion of the binary search (ϵ) is given by the user. Please refer to the extended version of the paper [4] for a low-complexity technique (a binary search among stretch-intervals) to compute the optimal maximum stretch.

Suppose that we are given $\epsilon > 0$. The binary search is conducted using Algorithm 1. This algorithm allows to approach the optimal stretch, as stated by the following theorem (the proof can be found in [4]).

Theorem 2. *For any $\epsilon > 0$, Algorithm 1 computes a stretch $\mathcal{S}$ such that there exists a schedule achieving $\mathcal{S}$ and $\mathcal{S} \leq \mathcal{S}_{\text{opt}} + \epsilon$, where $\mathcal{S}_{\text{opt}}$ is the optimal stretch. The complexity of Algorithm 1 is $O(\log \frac{\mathcal{S}_{\text{max}}}{\epsilon})$.*

Algorithm 1: Binary search

```
begin
    $\mathcal{S}_{\text{inf}} \leftarrow 1$
    $\mathcal{S}_{\text{sup}} \leftarrow \mathcal{S}_{\text{max}}$
    while $\mathcal{S}_{\text{sup}} - \mathcal{S}_{\text{inf}} > \epsilon$ do
        $\mathcal{S} \leftarrow (\mathcal{S}_{\text{sup}} + \mathcal{S}_{\text{inf}})/2$
        if Polyhedron (K) is empty then
            $\mathcal{S}_{\text{inf}} \leftarrow \mathcal{S}$
        else
            $\mathcal{S}_{\text{sup}} \leftarrow \mathcal{S}$
    return $\mathcal{S}_{\text{sup}}$
end
```

3.2. Online setting

Because we target an online framework, the scheduling policy needs to be modified upon the completion of an application, or upon the arrival of a new one. The idea is to make use of our study of the offline case. However, we cannot pretend to optimality any longer as we now have only limited information on the applications.

When a new application $A_{k_{\text{new}}}$ arrives at time $T_{\text{new}} = r^{(k_{\text{new}})}$, we consider the applications $A_0, \ldots, A_{k_{\text{new}}-1}$, released before T_{new}. We call $\Pi_{\text{rem}}^{(k)}$ the (fractional) number of tasks of application A_k remaining at the master at time T_{new}. For sake of simplicity, we do not consider the applications that are totally processed, and we thus have $\Pi_{\text{rem}}^{(k)} \neq 0$ for all applications. For the new application, we have $\Pi_{\text{rem}}^{(k_{\text{new}})} = \Pi^{(k_{\text{new}})}$. We also consider as parameters the state $B_u^{(k)}(t_{k_{\text{new}}})$ of the buffers at time T_{new}. We also have $B_u^{(k_{\text{new}})}(t_{k_{\text{new}}}) = 0$.

As previously, we compute the optimal max-stretch using Algorithm 1. For a given objective $\mathcal{S}$, we have a convex polyhedron defined by the linear constraints, which is non empty if and only if stretch $\mathcal{S}$ is achievable. The constraints are slightly modified in order to fit the online context. First, we recompute the deadlines of the applications: $d^{(k)} = r^{(k)} + \mathcal{S} \times MS^{*(k)}$. Note that now, all release dates are smaller than T_{new}, and all deadlines are larger than T_{new}. We sort the deadlines by increasing order, and denote by t_j the set of ordered deadlines: $\{t_j\} = \{d^{(k)}\} \cup \{T_{\text{new}}\}$ such that $t_j \leq t_{j+1}$. The constraints are the same as the ones used for Polyhedron (K), except the constraint on the number of task processed, which is updated to account for the remaining number of tasks to be processed.

As described for the offline setting, a binary search allows to find the optimal max-stretch. Note that this "optimality" concerns only the time interval $[T_{\text{new}}, +\infty]$, assuming that no other application will be released after T_{new}. This assumption will not hold true in general, hence our schedule will be suboptimal. The stretch achieved for the whole application set is bounded by the maximum of the stretches obtained by the binary search each time a new application is released.

4. MPI experiments and SimGrid simulations

We have conducted several experiments in order to compare different scheduling strategies, and to show the benefits of the algorithms presented in this work.

As our fluid model requires a total control of the rate of computation and communication, it is quite hard to implement in practice. During the experiments we used the *one-port* model for the communication, which serializes sending from the master, and we assume that a worker has to completely receive a task before starting its computation, and that it cannot perform several computations simultaneously. Please refer to [4] to see how optimality results can be translated under this more constraint model.

We first present the heuristics. Then we detail the platforms and applications used for the experiments. Finally, we expose and comment the numerical results.

The code and the experimental results can be downloaded from: `http://graal.ens-lyon.fr/~lmarchal/cbs3m/`.

4.1. Heuristics

In this section, we present strategies that are able to schedule multi-applications in an online setting. Although far from the optimal in a number of cases, such strategies are representative of existing Grid schedulers. We compare sixteen algorithms in the experiments. First we outline policies for selecting the set of applications to be executed:
• **FIFO** (First In First Out): applications are computed in the order of their release dates.
• **SPT** (Shortest Processing Time): released applications are sorted by non-decreasing processing time (which is approximated by MS^*). The first application must be completed before we determine the next one to be executed.
• **SRPT** (Shortest Remaining Processing Time): at each release date, released applications are sorted by non-decreasing processing time, according to the tasks that remains to be scheduled, and the applications are fully executed one after the other in this order until a new release date occurs.
• **SWRPT** (Shortest Weighted Remaining Processing Time): it is very similar to **SRPT**, but the remaining processing time of the released applications are weighted with MS^*. In practice, it gives small applications a priority against large applications which are almost finished.

Next we describe policies for resource selection:
• **RR** (Round-Robin): all workers are selected in a cyclic way.
• **MCT** (Minimum Completion Time): given a task of an application, it selects the worker which will finish this task first, given the current load of the platform.
• **DD** (Demand-Driven): workers are themselves asking for a task to compute as soon as they become idle.

The four application selection policies and the three ressource selection rules lead to twelve different greedy algorithms. We also test a more sophisticated algorithm:
• **MWMA** (Master Worker Multi-Applications): this algorithm computes on each time interval a steady-state strategy to schedule the available applications, as presented in [3]. All available applications are running at the same time, and each application is given a different fraction of the platform according to its weight. This weight can be derived from either the remaining number of tasks of the applications (variant called **NBT**), or the remaining time of computation of the applications (variant called **MS**). Both variants are compared in the experiments.

Finally, there is the strategy presented in this paper, called **CBS3M** (Clever Burst Steady-State Stretch Minimization). We test it with two variants, both a FIFO or EDF (Earliest Deadline First) policy for the workers to choose the next task to compute among those they have received. Both the **CBS3M** and the **MWMA** strategies make use of linear programs to compute their schedule. These linear programs are solved using glpk, the Gnu Linear Programming Kit [8].

4.2. Platforms

In this section, we conduct experiments on a real platform, in order to have an insight of the behavior of the algorithms. We also run multiple simulations, in order to get more results about their performance.

Experimental setting. Experiments were conducted on a cluster composed of nine processors. The master is a SuperMicro server 6013PI, with a P4 Xeon 2.4 GHz processor, and the workers are all SuperMicro servers 5013-GM, with P4 2.4 GHz processors. All nodes have 1 GB of memory and are running Linux. They are connected with a switched 10 Mbps Fast Ethernet network. In order to artificially enhance its heterogeneity, we slow down some communication links, by multiplying the size of messages targeting given workers by a constant factor. Similarly, we slow down some processor speeds by performing tasks several time on given workers. The experiments are performed using the MPICH-2 communication library [9].

We create ten different fully heterogeneous platforms. The communication and computation slowdowns are uniformly chosen between 1 to 10.

Simulation setting. An extensive set of simulations is performed using SimGrid [12]. The parameters of the simulated platforms are kept as close as possible to the actual experimental framework so that simulations can be considered a direct complement of the experimental MPI setting. In a first step, we run the exact same experiments (with the same platform configuration and application scenario) to make sure that our simulation behaves similarly to the MPI experiments. Then, we conduct an extensive set of simulations with larger applications.

4.3. Applications

For our experiments and simulations, we randomly generated the applications parameters, with the following constraints in order to be realistic:

1. the release dates of the applications follow a log-normal distribution as suggested in [7];
2. the total amount of communications and computations for an application is randomly chosen with a log-normal distribution between realistic bounds, and then split into tasks.

The number of tasks for one application is bounded above by the minimum amount of communication and computation allowed for one task. The parameters used in the generation of the applications for the experiments and the simulations are described in [4].

4.4. Results

In this section we describe the results obtained on all different platforms, experimental or simulated.

Experimental results. We express the performance of any given algorithm on one problem instance as the ratio of the max-stretch obtained by the algorithm on this instance over the theoretical optimal max-stretch obtained by linear programming.

The full results of the experiments can be found in [4]. Figure 2 summarizes the experiments for the best four algorithms, **CBS3M** using EDF policy, in both the offline and online versions, **MWMA NBT** and **SWRPT**.

Our results show that **CBS3M** achieve far better performance than any other strategy in all but two experiments. Surprisingly, the offline version is not always better than the online version. The offline version knows the future and thus should achieve better performance. However, it suffers from discrepancies between the actual characteristics of the platform and those of the platform model. The online version is able to circumvent this problem as it takes into account the work effectively processed to recompute the schedule at each new application arrival. This gain of reactivity compensates for the loss due to the lack of knowledge of the future.

The **MWMA** algorithms lie in between our algorithms and the greedy strategies.

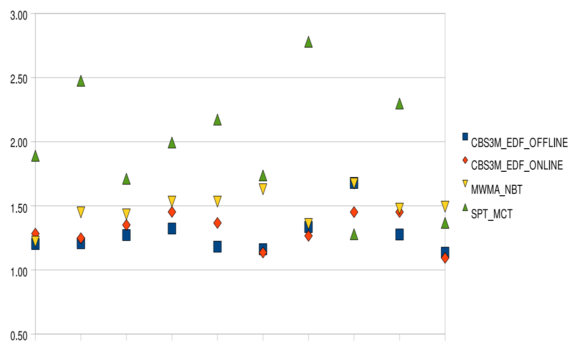

Figure 2. Relative max-stretch of best four heuristics in the MPI experiments.

Simulation on experimental platform. We have run exactly the same experiments with simulations, using the same platform configurations and application scenarios. We compare the difference between the relative max-stretch in both cases. As a result, we found that the average difference on the relative max-stretch is around 21%, with a standard deviation of 57%. These results show that our simulations are generally close to those obtained on a real platform.

Simulation results. In this section, we run 1000 experiments The results of all heuristics for the max-stretch metric can be found in [4]. Figure 3 shows the evolution of some heuristics (the best ones) over the load of the scenario. The **CBS3M** heuristics perform very well for the max-stretch: **CBS3M** EDF ONLINE achieves the optimal max-stretch in 65.2% of the experiments. This heuristic achieves great performance, with an average max-stretch of 1.16 times the optimal max-stretch, and a worst case of 1.93. In the extended version of the paper [4], we detailed results of the different heuristics for a bunch of metrics.

The good results of the **CBS3M** heuristics can be explained with the fact that they make very good use of the platform, by scheduling simultaneously several applications when it is possible, for example when the communication medium has still some free bandwidth after scheduling the most critical application. All other heuristics (except **MWMA**) are limited to scheduling only one application at a time, leading to an overall bad utilization of the computing platform.

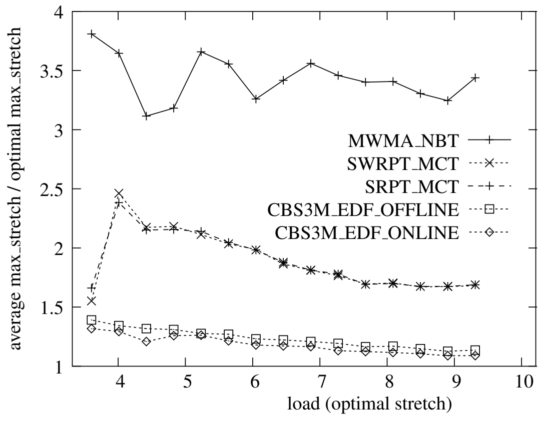

Figure 3. Evolution of the relative max-stretch of best heuristics in the simulations under different load conditions.

5. Conclusion

In this paper, we have studied the problem of scheduling multiple applications, made of collections of independent and identical tasks, on a heterogeneous master-worker platform. Applications have different release dates. We aimed at minimizing the maximum stretch, or equivalently at minimizing the largest relative slowdown of each application due to their concurrent execution. We derived an optimal algorithm for the off-line setting (when all application sizes and release dates are known beforehand). We have adapted this algorithm to an online scenario, so that it can react when new applications are released.

We have compared our new algorithms against classical greedy heuristics, and also against some involved static multi-applications strategies. Experiments were run both on a real cluster, using MPI, and through extensive simulations, conducted with SimGrid. Both experimental comparisons show a great improvement when using our **CBS3M** strategy, which achieves an averaged worse max-stretch only 16% greater than the off-line optimal max-stretch. To the best of our knowledge, this work is the first attempt to provide efficient scheduling techniques for multiple bags-of-tasks in an online scenario.

Future work includes extending the approach to other communication models and to more general platforms (such as multi-level trees). It would also be very interesting to deal with more complex application types, such as pipeline or even general DAGs.

References

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal sharing of bags of tasks in heterogeneous clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.

[2] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distributed Systems*, 15(4):319–330, 2004.

[3] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *International Parallel and Distributed Processing Symposium IPDPS'2006*. IEEE Computer Society Press, 2006.

[4] A. Benoit, L. Marchal, J.-F. Pineau, Y. Robert, and F. Vivien. Offline and online scheduling of bag-of-tasks applications on heterogeneous platforms. Research Report 2007-48, LIP, ENS Lyon, France, Dec. 2007.

[5] BOINC: Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu`.

[6] H. Casanova and F. Berman. Grid'2002. In F. Berman, G. Fox, and T. Hey, editors, *Parameter sweeps on the grid with APST*. Wiley, 2002.

[7] D. G. Feitelson. *Workload Characterization and Modeling Book*. electronic draft, no published yet.

[8] GLPK: GNU Linear Programming Kit. `http://www.gnu.org/software/glpk/`.

[9] W. Gropp. Mpich2: A new start for mpi implementations. In *PVM/MPI*, page 7, 2002.

[10] B. Hong and V. Prasanna. Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*. IEEE Computer Society Press, 2004.

[11] H. J. Kim. A novel optimal load distribution algorithm for divisible loads. *Cluster Computing*, 6(1):41–46, 2003.

[12] A. Legrand, L.Marchal, and H. Casanova. Scheduling Distributed Applications: The SIMGRID Simulation Framework. In *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)*, pages 138–145, May 2003.