# Dynamic Scheduling of Independent Tasks with a Budget Constraint on a Desktop Grid

Ahmed Elleuch[1], Lilia Chourou[2], and Mohamed Jemni[2]

[1] University of Manouba, National School of Computer Sciences, CRISTAL Laboratory, Tunisia,
ahmed.elleuch@ensi.rnu.tn
[2] University of Tunis, Higher School of Sciences and Techniques of Tunis, LaTICE Laboratory, Tunisia
lilia.gherir@planet.tn, mohamed.jemni@fst.rnu.tn

**Abstract.** In an Internet desktop grid, the computing power may be leveraged by providing economic incentive for computer owners. Using batch mode scheduling heuristics, such as MinMin, MaxMin or Sufferage, this paper describes an algorithm that performs independent task scheduling on a large scale heterogeneous system, with a budget constraint. Under the restriction of never exceeding a periodically refreshed budget, successive optimizations are performed to enhance the makespan by finding a suitable distribution of the budget among the given tasks, through the given time horizon. For the scalability purpose, the computing resources are partitionned according to their speed into classes. First, a schedule on these classes is computed using one of the mentioned well-known heuristics, and then, tasks assigned to a class are actually scheduled to the respective computing resources, as these tasks were ranked by the first scheduling algorithm. To evaluate the effectiveness of our scheduling approach, we have built a simulator that uses real traces obtained from the BOINC based XtremLab project. The results show that the repartition of the budget among tasks, and through time, improves the makespan by a significant factor. They also show that the adaptation to the grid scale can achieve comparable makespan when the availability interval of computing node is not too small.

**Keywords:** independent task scheduling, batch mode, high throughput computing, budget constraint, large scale heterogeneous computing

## 1 Introduction

Over the past few years we have seen an increasing use of commodity technologies for high-performance computing. The supercomputer vendors are using more and more common commodity components and processors. Gone are the days when high performance computing was possible only for those who could afford expensive supercomputers. Nowadays, very large scientific applications in genome analysis, molecular interaction, protein modeling, etc, are being developed using middleware technologies that aggregate the computing power of an extensive number of Internet-connected machines [1, 3, 11, 14]. To support this expanding interest and acceptance of the use of commodity computers to solve large-scale computational problems, desktop grids are being developed as promising high-performance and high-throughput computing infrastructures.

For peer-to-peer and global computing systems, it is rather common that volunteers make a commitment to donate unused computational resources for public interest, for challenge, for fame or simply just for amusement. It is through the introduction of economic incentives and by embedding a pricing strategy for welfare that it would be possible to increase the number of computing resource providers as well as the number of consumers. This new approach would allow providers' investments and consumers' expenses to be better justified. Providers will be able to address wider markets and consumers will not have to continuously maintain and adapt their computing infrastructures to evolving needs and will pay only for what they really use. Computing would be a commodity as replacement of computer and software goods. The emergence and recent development of cloud computing platforms shall extend even further shifting of computing to computing centers connected to the Internet [9]. However, until now, cloud computing services cannot be regarded as real commodities because these services are not fully fungible and cloud providers have their own pricing schemes.

Task scheduling is one of the most challenging problems facing Internet computing. As the worldwide Internet network is too slow to support tightly coupled applications, in the present work, we consider only independent task scheduling. Such scheduling problem, on non-identical processors, is well known to be NP-complete [18]. Many distributed and parallel computing researches have explored heuristics for independent task scheduling that aim to maximize execution throughput on heterogeneous systems [18, 17, 21, 4]. However, little work has considered economic incentive criteria in the objective function. The present work describes an adaptation method of classical batch mode scheduling heuristics to support independent task scheduling with a budget constraint on heterogeneous Internet-connected machines. Under the restriction of never exceeding a periodically refreshed budget, the goal is to minimize the makespan. Besides scheduling, we must also address other issues including security, quality of service guarantees, pricing, accounting and trusted payment. These issues are not covered in the present paper but are investigated in other ongoing works. Scheduling independent tasks with economic constraints on a set of available heterogeneous computational resources is the challenging problem addressed in this paper.

Most widely known Internet computing projects, such as BOINC based projects [3], Entropia [11] and XtremWeb [14], described as peer-to-peer, actually rely on a non-peer central element. The central element is used for global scheduling, task initialization, maintaining a central repository for machine and task information, collection of results, etc. The other nodes, called the peers, perform the actual computations. This centralized approach makes the computation of schedules easier. In line with this same approach we aim at adapting the classical batch mode scheduling heuristics to performance and budget constraints.

The considered heuristics are MinMin, MaxMin [17] and Sufferage [21] ; they are based on centralized computation of schedules. As the time complexity of these heuristics depends on the number of workers in the grid, these heuristics are not well suited for large-scale systems. To overcome this drawback, we have partitioned the workers into classes according to their speed. First, a schedule on these classes is computed using one of the mentioned well-known heuristics, and then, tasks assigned to a class are actually scheduled to the respective workers as these tasks were ranked by the first scheduling algorithm.

Desktops and workstations represent a big underutilized reserve of heterogeneous resources [2]. Most components of these computers spend a great part of their time doing nothing. Especially, it is the case of CPUs, even if you are actively working away on your computer, your CPU is likely to be waiting for keystrokes, mouse clicks and so on. However, you may also run a CPU intensive application. Besides, your computing node may join or leave the grid at any moment. Clearly, the processing power available for the Internet computing usage is highly variable, a scheduler has no guarantees regarding the availability and the computing speed of the execution nodes. The proposed scheduling method is designed to support scheduling on such non-dedicated workers.

The paper is organized as follows. Section 2 discusses the related work. Section 3 gives the outline of our scheduling model and provides a comprehensive description of the considered classical scheduling algorithms. Section 4 describes the proposed scheduling algorithm. Section 5 gives the experimental environment, and presents the simulation-based experimental results. Finally, Section 6 concludes our work and presents the future work.

## 2 Related Work

Ibarra and Kim [18] have described several heuristic-based algorithms for task scheduling on non-identical processors. Two of them have been implemented by Freund et al. as potential dynamic batch mode scheduling algorithms in the SmartNet system [17]. They called them MinMin and MaxMin algorithms. Another well-known heuristic-based algorithm is the Sufferage algorithm described in [21]. A short description of these heuristics is given in the section 3.4. In this last work, Maheswaran et al. have also compared the above three batch mode scheduling algorithms and several immediate mode scheduling algorithms. They have considered makespan as a measure of the throughput of the computing system. They showed that for heterogeneous computing, when there are a large number of tasks and high task arrival rate, the batch mode is able to provide a smaller makespan than the immediate mode. Indeed, these conditions will insure that there will be a sufficient number of tasks in order to take better decisions and to keep the workers busy between mapping events and while a mapping is being computed [21]. MinMin, MaxMin and Sufferage have been used in many grid execution environments and especially in AppLeS [10] and GrADS [12]. The AppLeS (Application-Level Scheduling) template uses a specific scheduler within each application that could use one of these heuristics. GrADS (Grid Application Development Software) provides a generic application-level scheduler that maps workflow application tasks to a set of resources according to the heuristic that compute a schedule with the minimum makespan. As MinMin, MaxMin and Sufferage are becoming benchmarks, we are looking to adapt them to support high throughput computing through economic incentive-based scheduling onto an Internet Desktop Grid. While these heuristics have been proposed for mono-objective optimization, Phatanapherom and Uthayopas have used them for multi-objective optimization [22]. Instead of makespan, they have defined another objective function which is a weighted-sum of execution time, ready time, cost, accumulative cost and time period between completion time and deadline. Using this function, new economic scheduling algorithms have been proposed as an extension to conventional scheduling algorithms (MinMin, MaxMin, Sufferage). These extended algorithms optimize rental cost of running tasks but are not able to support optimization strategies that must respect a budget limit.

Buyya et al. have designed a GRid Architecture for Computational Economy (GRACE) [5, 6] which corresponds to a system-level middleware infrastructure that includes a user-level broker, called Nimrod/G, and a trading mechanisms working closely with the Globus grid middleware [16]. As a part of the Nimrod/G broker, three budget and deadline constrained scheduling algorithms are proposed [7]. These algorithms differ in that: the first one tries to minimize the time, the second one tries to minimize the cost while the third one does not minimize either. In [8], a forth algorithm was proposed, called cost–time optimization, which keeps the cost at a minimum and applies time-optimization while allocating tasks to a group of equal-cost resources. However, all these algorithms assign tasks in an arbitrary order without optimizing the scheduling by taking into account the expected completion times of tasks on computing resources, as it is done in MinMin, MaxMin or Sufferage. Besides, all the mentioned budget constrained scheduling algorithms are application-centric. The submitted tasks are supposed to belong to a same application which has its specific deadline and budget limits. Another interesting case study, which is our main concern here, is the system-centric scheduling constrained by a budget shared among different users. Both the computing power and the budget are considered as shared resources. The corresponding model is detailed in the section 3. Under the restriction of never exceeding the budget, we have optimized the makespan by finding a suited repartition of the limited - but periodically recharged - budget among tasks and through time. As we are looking to support high throughput computing, our proposed generic scheduling algorithm has no deadline constraints.

The MinMin, MaxMin and Sufferage scheduling algorithms operate in a centralized fashion and are not designed to support a massive number of workers: to schedule a set of tasks onto a grid the time complexity is $O(t^2 * w)$, where $t$ is the number of tasks and $w$ is the number of workers in the grid (for Sufferage, this complexity is in the worst case) [21]. To reduce the time complexity of the scheduling algorithm, in [13] and as a part of GrADS framework, a search space is pruned. Resources are grouped into disjoint subsets such that network delays within each subset are lower than network delays between subsets. In [23], Venugopal and Buyya have proposed a scheduling algorithm of data intensive applications using a detailed cost model associated with the movement and the processing of datasets. A search space is also limited by selecting computing resources from a sorted list and by making assignments in an incremental fashion. In [26], a guided search procedure was applied using a genetic algorithm to compute a static scheduling that minimizes the execution time of a workflow while meeting user's budget constraint. In the present work, tasks are independent and are not known a priori. Only dynamic scheduling heuristics are considered. We do not address overheads due to file staging and so we do not apply an approach that limits the search space in a similar way as in [13] or [23]. To reduce a search space, which is enlarged by a massive number of computing workers, the workers are gathered into classes of equivalent workers. Using the MinMin, the MaxMin or the Sufferage heuristic, a first mapping to these classes is computed. Then final task schedules onto actual workers are computed. An information server is used to dynamically lookup the computing workers in the grid and to determine their average speeds.

## 3 Grid Scheduling Model

We consider the scenario where a computing center aims to provide a high throughput computing service to its users not necessarily by acquiring supercomputers, but by appealing to grid resource providers which receive, in exchange, a monetary reward. A computing center has its own scheduler and a global periodically refreshed budget shared among all its users, users submit tasks to the same scheduler. We do not address starvation problem where a task may wait a long time before it starts. To resolve the starvation problem, in the same way as described in [21], aging schemes may be used.

### 3.1 Task Scheduling Model

All tasks are assumed to be independent and have no deadlines or priorities associated with them. They have a coarse granularity. The delay needed to start-up a remote task, including task transfer delays, is assumed to be negligible as compared to task execution time. A rule of thumb is: if a remote execution of a task needs more time to ship data and code than the computing time, then it is cheaper to use local computing resources. So, we suppose that such a task is never submitted to our grid-wide scheduler. Besides, the running time to compute task scheduling must be negligibly small compared with the average task execution time. Especially this last condition must hold even when there is a high number of workers. Task arrival times may be random and are unknown in advance. A dynamic non-preemptive scheduling scheme is applied using the batch mode. Tasks are not scheduled immediately but they are periodically collected into a set, called a meta-task, and then scheduled at the end of each period. The scheduling period is denoted $R$. A meta-task includes newly (not yet scheduled) arrived tasks and those that were scheduled in earlier scheduling events but did not began or they failed to complete their execution. Indeed, a task execution may be aborted due to a network or a worker failure. At the $k^{th}$-scheduling event, the meta-task is denoted $M^k$ and the wall-clock time is denoted $D^k$.

A parametric function is used to estimate a task completion time. Each task $t_i$ is characterized by its length $l_i$, known in advance and used in the parametric function. This length may be a loop iteration count, an input data length or whatever parameter needed to estimate the task execution time. We assume that all tasks have the same unit of length that we call operation. For simplicity, and without loss of generality, the parametric function used to estimate the expected completion time of a task $t_i$ on a worker $w_j$ (at $k^{th}$-scheduling event) is given by (1).

$$ECT^k(t_i, w_j) = r_j^k + \frac{l_i}{s_j^k} \tag{1}$$

where $s_j^k$ is the average speed of $w_j$, $r_j^k$ is the maximum of $(D^k, r_j)$, $r_j$ denotes the expected time at which $w_j$ will become ready to execute a new task, after finishing the execution of any previously assigned task. $r_j$ is updated according to the effective progress of the last submitted task at the previous scheduling event *k-1*. Note that at the $k^{th}$-scheduling event, if the worker $w_j$ is ready before the scheduling event time $D^k$ then the task $t_i$ could not be started before $D^k$.

### 3.2 Grid Model

Computational grids include a variable number of computers distributed over the Internet. Computers are heterogeneous and have various nominal computing powers. Each computer is called a worker and may offer to the grid its remaining computing power - which is not used by the local users (i.e. owners) - or a dedicated fraction of its processing capacity. The resulting available computing power is modeled as an execution speed and is expressed as the number of operations computed per time unit. The speed of each worker varies over time according to the load generated by its local users. This execution speed model is used to represent the computer heterogeneity and the computing power availability. As almost all workers are time-sharing systems and since the submitted tasks are CPU intensive ones, the pseudo-parallel execution of two tasks may take more time than executing them sequentially. So, on each worker is launched only one non-local task at a time per core.

Any worker may leave (respectively, join) the grid at any moment. This may happen due to an owner decision or simply as consequence of a worker or network failure (respectively, end of failure). A worker that leaves and then joins again the grid is considered as a new worker. At the $k^{th}$-scheduling event, the set of workers is denoted $G^k$. An information service, such as the directory service discribed in [15], is used to lookup the workers that belong to $G^k$. All workers must be registered at this server. A prediction mechanism is also needed to determine the expected average speed of each worker. The Network Weather Service (NWS) [24] may be used for this purpose. In this paper we are not concerned with how prediction is done. At the $k^{th}$-scheduling event, the set of workers with an average speed $s_n$ is denoted $G^k_{s_n}$. We assume that there are $N$ representative standard values of the average speeds. While the number of workers $|G^k|$ may be massive, the number of average speeds $N$ is assumed to be reduced. For the considered heuristics, it is possible to explore iteratively all the $N$ alternative average speeds in a reasonable time. For an average speed $s_n$, $|G^k_{s_n}|$ may be massive (note that $|G^k|$ is equal to the sum of all $|G^k_{s_n}|$, $1 \leq n \leq N$).

### 3.3 The Budget Model

The execution of a task is budget constrained. The available budget must never be exceeded. As in [19], we assume that the budget is periodically recharged. The recharge value is denoted $B$. At each scheduling event, a scheduler can never accumulate more than $B$ credits. This automatic refresh of the budget avoids hoarding and starvation.

Any node is able to execute every task, the rental cost of running a task $t_i$ on worker $w_j$ with an average speed $s_n$ is:

$$RentalCost(t_i, s_n) = l_i * p^k_{s_n} \tag{2}$$

where $p^k_{s_n}$ is the execution price of an operation unit. At each scheduling event $k$, each operation price $p^k_{s_n}$, $1 \leq n \leq N$, is obtained from the information server. The pricing algorithm is studied in another ongoing work. Note that if the execution of a task is aborted then no penalty charges are incurred.

### 3.4 The General Scheduling Algorithm

The classical scheduling heuristics MinMin, MaxMin and Sufferage have the same general algorithm [10]. The pseudo-code of this algorithm is depicted in in Figure 1. The set of not yet scheduled tasks is denoted by $\underline{M}^k$. Tasks are assigned iteratively to workers. For each task $t_i$, a selection metric $d_i$ is computed as a function of the expected completion times of the task $t_i$ on all workers. When all the metrics - for the not yet scheduled tasks in $\underline{M}^k$ - are computed, they are used to select a best task $t_p$. For this task, the worker $w_q$ that gives the minimum expected completion time $c_{p,q}$ is determined. The task $t_p$ is then scheduled on the worker $w_q$. This process is repeated until all tasks are scheduled. The functions *Metric* and *BestMetric* entirely define a heuristic as shown in Table 1.

### 3.5 Notation Summary

Table 2 gives a summary of the notation we have just used in the section 3.

## 4 The Adapted Algorithm

### 4.1 Adaptation to the budget constraint

With a slight modification of the algorithm described in Figure 1, we could respect the budget limit as follows. At each iteration and when selecting a new

```
At each scheduling event $k$
 1: $\underline{M}^k = M^k$
 2: while $\underline{M}^k \neq \emptyset$
 3:     for each $t_i$ in $\underline{M}^k$
 4:         for each $w_j$ in $G^k$
 5:             $c_{i,j} = ECT^k(t_i, w_j)$
 6:         $d_i = Metric_{w_j \in G^k}(c_{i,j})$
 7:     end for
 8:     select $p$ such as $d_p = BestMetric_{t_i \in \underline{M}^k}(d_i)$
 9:     select $q$ such as $c_{p,q} = Minimum_{w_k \in G^k}(ECT^k(t_p, w_k))$
10:     schedule task $t_p$ on $w_q$
11:     remove $t_p$from $\underline{M}^k$
12: end while
```

**Fig. 1.** The general scheduling algorithm

**Table 1.** Definition of *Metric* and *BestMetric* functions for the classical heuristics

| ***Heuristic*** | ***Metric*** | ***BestMetric*** |
|---|---|---|
| ***MinMin*** | Minimum | Minimum |
| ***MaxMin*** | Minimum | Maximum |
| ***Sufferage*** | Difference between second minimum and minimum | Maximum |

**Table 2.** Notation summary

| ***Symbol*** | ***Description*** |
|---|---|
| $D^k$ | The the wall-clock time at the $k^{th}$-scheduling event |
| $R$ | The scheduling period |
| $M^k$ | The meta-task at the $k^{th}$-scheduling event |
| $t_i$ | A task $i$ |
| $l_i$ | The length of the task $t_i$ |
| $G^k$ | The set of workers at the $k^{th}$-scheduling event |
| $w_j$ | A worker $j$ |
| $r_j$ | The expected ready time of the worker $w_j$ |
| $r_j^k$ | $r_j$ or at least $D^k$ |
| $s_j^k$ | The average speed of worker $w_j$ at the $k^{th}$-scheduling event |
| $N$ | The number of representative values of the average speeds |
| $G_{s_n}^k$ | The set of workers with an average speed $s_n$ at the $k^{th}$-scheduling event |
| $B$ | The recharge value of the budget at each scheduling event |
| $p_{s_n}^k$ | The execution price of an operation unit by a worker with an average speed $s_n$ |

worker-task match, the metric associated to a task $t_i$ is evaluated using only the workers that do not need a cost which exceeds the expected remaining budget $\underline{B}$. When the budget limit is exhausted, the not yet scheduled tasks are delayed to the next meta-task. Figure 2 gives a sketch of this first solution.

```
At each scheduling event $k$
1: $\underline{M}^k = M^k$
2: $\underline{B} = B$
3: while $(\underline{M}^k \neq \emptyset)$ and $(\underline{B} \neq 0)$
4:   for each $t_i$ in $\underline{M}^k$
5:       $S_i^k = \{n : 1 \leq n \leq N\}$
6:       for $n$=1 to $N$
7:           if $RentalCost(t_i, s_n) > \underline{B}$ then
8:               remove $n$ from $S_i^k$
9:       $W_i^k = \bigcup_{n \in S_i^k} G_{s_n}^k$
10:      for each $w_j$ in $W_i^k$
11:          $c_{i,j} = ECT^k(t_i, w_j)$
12:      if $W_i^k = \emptyset$ then
13:          remove $t_i$ from $\underline{M}^k$
14:      else
15:          $d_i = Metric_{w_j \in W_i^k}(c_{i,j})$
16:   end for
17:   if $\underline{M}^k \neq \emptyset$ then
18:       select $p$ such as $d_p = BestMetric_{t_i \in \underline{M}^k}(d_i)$
19:       select $q$ such as $c_{p,q} = Minimum_{w_k \in W_p^k}(ECT^k(t_p, w_k))$
20:      schedule task $t_p$ on $w_q$
21:      remove $t_p$ from $\underline{M}^k$
22:      $\underline{B} = \underline{B}$-$RentalCost(t_p, s_{w_q})$ //$s_{w_q}$ is the speed of $w_q$
23: end while
```

**Fig. 2.** The non-optimized version of the budget constrained scheduling algorithm

However, all the considered heuristics tend to select the fastest, and likely more expensive, workers. Such selection increases the expenditure and risks to quickly exhaust the budget. Clearly, this situation does not optimize neither the makespan nor the use of the budget. Indeed, it may be wasteful to minimize the execution time of some tasks and then wait the refresh of the budget to schedule the other tasks. The delayed tasks will constitute a supplementary load for the next meta-task. On the other hand, it could be possible to use less expensive workers and, with the saved expenses, schedule much more tasks and reduce the overall makespan. From this discussion emerges the importance of a good repartition of the budget among tasks and through time. To be able to schedule all the tasks of a meta-task $M^k$ with the budget $B$, we should select workers with an operation unit price close to:

$$\frac{B}{\sum_{t_m \in M^k} l_m} \tag{3}$$

For this purpose, a task $t_i$ could be scheduled on a worker only if the following condition is verified:

$$\frac{\overline{B} + l_i * p^k_{s_n}}{\sum_{t_m \in \overline{M}^k} l_m + l_i} \leq \frac{B}{\sum_{t_m \in M^k} l_m} \tag{4}$$

where $\overline{B}$ denotes the already allocated budget ($\overline{B} = B - \underline{B}$), $\overline{M}^k$ denotes the subset of scheduled tasks at the iteration $k$ ($\overline{M}^k \subset M^k$) and $s_n$ is the average speed of the target worker.

Besides, the budget could be recharged before the actual start of all tasks in the meta-task $M^k$. The not yet started tasks would be rescheduled at the iteration *k+1* according to a newly updated budget. However, according to (4), these tasks have been taken into account in the repartition of the $k^{th}$-budget among tasks while they should not be considered. This potential imprecision may be removed by adapting the time repartition of allocated expenses. To not underutilize the budget $B$ during the refresh period $R$, allocated expenses through time should be close to $B/R$. If we consider a current task to be potentially scheduled on worker $w_j$ then we should have:

$$\frac{\overline{B}}{r^k_j - D^k} \approx \frac{B}{R} \quad if\ r^k_j > D^k$$

recall that $D^k$ denotes the wall-clock time of the $k^{th}$-scheduling event and $r^k_j$ denotes the expected ready time of the worker $w_j$. Note that we do not have to consider workers with a ready time greater than $D^k+R$. When $D^k$ is greater or equal to $r^k_j$, it is still the beginning of the period R, so the budget is not underutilized and the condition (4) is sufficient. However, if $r^k_j$ is greater than $D^k$ and we have:

$$\frac{\overline{B}}{r^k_j - D^k} << \frac{B}{R}$$

then this situation leads to the creation of savings, denoted by $E^k_j$, given by the following expression:

$$E^k_j = \left(\frac{B}{R} - \frac{\overline{B}}{r^k_j - D^k}\right) * (r^k_j - D^k) = \frac{B * (r^k_j - D^k) - \overline{B} * R}{R}$$

Using this savings and applying the condition (4), a task $t_i$ could be scheduled on a worker $w_j$ (with an average speed $s_n$) that verifies the following condition:

$$\frac{\overline{B} - E^k_j + l_i * p^k_{s_n}}{\sum_{t_m \in \overline{M}^k} l_m + l_i} \leq \frac{B}{\sum_{t_m \in M^k} l_m}$$

or simply

$$\frac{\frac{B}{R} * (r_j^k - D^k) + l_i * p_{s_n}^k}{\sum_{t_m \in \overline{M}^k} l_m + l_i} \leq \frac{B}{\sum_{t_m \in M^k} l_m} \tag{5}$$

To take into account this condition in the adapted algorithm described by figure 2, we have modify the line 9 as follows:
$W_i^k = \{w_j : \exists n \in S_i^k, w_j \in G_{s_n}$, if $p_{s_n}$ is not the lowest price then
(if $r_j^k > D^k$ then condition (5) holds else condition (4) holds)$\}$

## 4.2 Adaptation to grid scale

The above general scheduling algorithm and the adapted one to the budget constraint are not well suited when there is a large number of workers. Indeed, at an iteration *k,* for each remaining task and on each worker that fulfills the budget constraint, we need to compute the expected completion time. The number of workers may be a massive number and then it is very time-consuming to compute separately all the expected completion times for all the potential workers. To overcome this problem, the ready time of a $s_n$ speed worker $w_j$ is globally approximated by (6):

$$r_j^k = r_{s_n}^k = \max\left(D^k, \tilde{r}_{s_n}^{k-1}\right) + \frac{\sum_{t_m \in \overline{M}_{G_{s_n}}^k} l_m}{|G_{s_n}^k| * s_n} \quad \textit{if all } G_{s_n}^k \textit{ workers are busy}$$

$$r_j^k = r_{s_n}^k = D^k \quad \textit{otherwise} \tag{6}$$

where $\tilde{r}_{s_n}^{k-1}$ is a real-time updated value of $r_{s_n}^{k-1}$ such as only the not-yet-finished tasks are taken into account ($k$>*0*), $r_{s_n}^0$ is the wall-clock time ($D^1$), $|G_{s_n}^k|$ is the number of $s_n$ speed workers and $\overline{M}_{G_{s_n}}^k$ is the current subset of tasks assigned to workers with an average speed $s_n$ at the scheduling event *k*. In the determination of $r_j^k$, the use of an arithmetic average time as an approximation of the execution duration of all scheduled tasks in $\overline{M}_{G_{s_n}}^k$ is justified by the fact that, anyway, the estimations are based on average speeds.

As a consequence of this global approximation of the ready times, and so also the expected completion times, the scheduling algorithm computes schedules of tasks onto a limited number $N$ of worker-sets ($G_{s_n}^k$, $1 \leq n \leq N$) without considering all the $|G^k|$ workers and assigning the tasks to the actual workers. The final adaptation of the general scheduling algorithm - to support both the budget constraint and a large number of workers - is obtained by making the following modifications to the algorithm described by figure 2. The notation of a worker $w_j$ has to be replaced by a speed index reference *n*, the ready time denoted by $r_j^k$ must replaced by $r_{s_n}^k$ as formulated by the expression (6), and the lines 9, 11, 15, 19 and 20 are respectively replaced by:

(9): $W_i^k = \{n : n \in S_i^k$, if $p_{s_n}$is not the lowest price then
(if $r_{s_n}^k > D^k$ then condition (5) holds else condition (4) holds)$\}$
(11): $c_{i,n} = r_{s_n}^k + \frac{l_i}{s_n}$

```
(15): $d_i = Metric_{n \in W_i^k}(c_{i,n})$
(19): select $q$ such as $c_{p,q} = \mathrm{Minimum}_{\mathrm{n} \in W_p^{\mathrm{k}}}(r_{s_n}^k + \frac{l_p}{s_n})$
(20): **Schedule** task $t_p$ **on** $G_{s_q}^k$
```

After this first scheduling, a second and final scheduling is needed to determine for each task the actual worker that will execute the task. This last scheduling is done as follows. When a worker $w_j$, with an average speed $s_n$, finishes the execution of its current task, the next task assigned to $G_{s_n}$(as it was ranked by the first scheduling algorithm), if any, is scheduled on the worker $w_j$.

## 5 Experimental results and discussion

In section 4, we have proposed successive enhancements to the general scheduling algorithm to support the budget constraint and to adapt the algorithm to the grid scale. The evaluation of the contribution of each of these optimizations is performed by simulation experiments using repeatable patterns of worker's and task's behavior. An essential issue in this simulation is the characterization of the heterogeneity and the availability of Internet desktop computers (i.e. workers). However, this characterization has been poorly understood [20]. Rather than to try to model the behavior of the workers, we used real traces obtained from XtremLab project [25]. The goal of this project is to monitor the availability of a large number of desktop PC's within a worldwide global computing infrastructure that uses the BOINC (Berkeley Open Infrastructure for Network Computing) platform [1].

XtremLab determines CPU availability using an active measurement method. To a monitored worker is submitted a task that iteratively computes integer and floating-point operations and periodically, nearly every 10 seconds, writes the number of iterations performed to a trace file. Each iteration corresponds to a known number of integer and floating point operations. A task length is expressed in iterations. For simplicity, these iterations are called operations. We structured the traces as a time series and used them to simulate the execution of tasks on workers. All the simulated tasks are assumed to make the same type of calculation and so to perceive availability such as that traced.

Several hundred of thousand active computers participate to BOINC projects but XtremLab project monitors a limited number of participating computers. In the present work, the used traces were collected between 02 March 2006 and 09 April 2007 from 4309 computers. Nevertheless, this not very large fraction of workers contains a variety of rather representative heterogeneous computers. Figure 3 shows the cumulative speed distribution of near $19{*}10^7$ measured speeds averaged over a period of 10 seconds (during 21939 CPU days). The average speed is $8.9{*}10^3$ operations/second; 90 % of measures corresponds to a speed less than $12{*}10^3$ operations/second. For higher speeds, the curve continues to always increase slightly until it reaches 100% which corresponds to a speed of $99{*}10^3$ operations/second. Hence, the speed values between 0 and $12{*}10^3$ are potentially more relevant than the upper values. The $N$ representative standard values of the average speed were chosen as follows. From 0 to 12*$10^3$ we selected 121 equidistant values (i.e., the integer interval between 0 and 120 multiplied by $10^2$ operations/second) and from 20*$10^3$ to 99*$10^3$ we selected 80 equidistant values (i.e., the integer interval between 20 and 99 multiplied by $10^3$ operations/second). A traced speed value is rounded to the nearest representative standard value. The half-integers are rounded to even numbers.

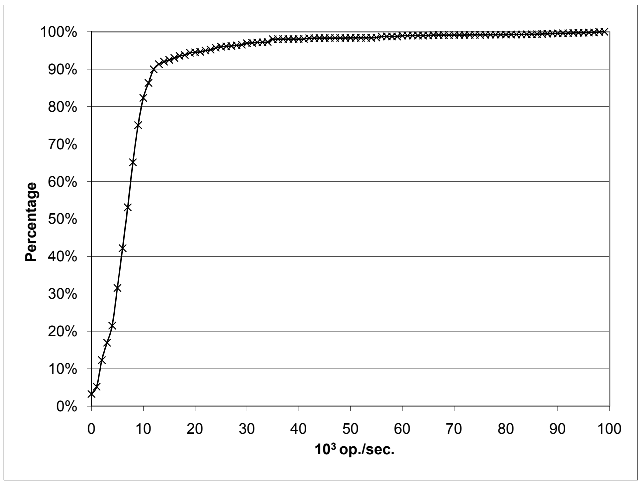


**Fig. 3.** Cumulative distribution of speeds

On a given worker and while it is available, the duration between two consecutive failures is called an availability interval. As XtremLab executes successive measurement tasks, called work unit, for each available processor, we have used an interpolation function to determine a presumable speed between two consecutive work units. Besides, XtremLab is not able to trace the precise date at which a worker becomes unavailable. We approximate this date by the end time of the last entirely executed work unit (before the worker becomes unavailable during a significant duration).

The simulated tasks are generated randomly: the size of an arriving task is a random value chosen between 5*$10^6$ ($l_{\min}$) and 12*$10^6$ ($l_{\max}$) operations. We have fixed a minimum length value of 5*$10^6$ operations because we assumed previously that tasks have a coarse granularity (see section 3.1). As 95% of the availability intervals reach 12*$10^6$ operations, this value has been chosen arbitrary as a maximum task length to limit the proportion of aborted tasks. Recall that the execution of a task must fit completely in an availability interval otherwise the task is aborted and rescheduled in the next meta-task.

For each month, in the trace period, we calculated the Average Number of available Workers (ANW) and the Average Cumulative Speed (ACS) of these available workers. We found that the ACS is not proportional to ANW. For instance, while for June and September we have not too different values of ANW, respectively 101 and 94 workers, the ACS of September is almost $12*10^6$ operations/second which is near the double of the ACS of June. These two months are very representative of many other months (except the beginning and the end months of the trace period). Without overloading the simulated desktop grid, we have chosen to tune and maximize the task arriving rate according to ANW and ACS of September. The so-generated task arrival patterns are used first, to carry out the simulation study using September worker time series and then, to simulate overloaded conditions, we have used the same task arrival patterns with June worker time series.

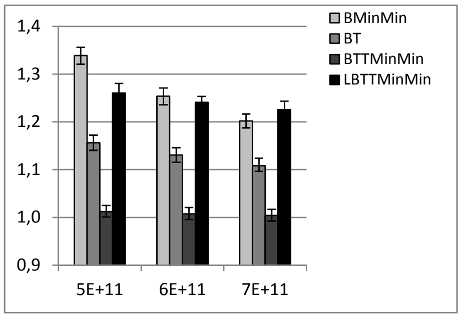
To make the average number of newly arriving tasks close to ANW of September, at each scheduling event, the number of arriving tasks has been randomly generated between 0 and 200. So, at each scheduling event, the consequent total average number of submitted operations is $85*10^7$ operations , not included operations that belong to the rescheduled tasks (i.e., $100*(l_{\min} + l_{\max})/2$). As the ACS of September is approximately $12*10^6$ operations/second, to compute $85*10^7$ operations, the grid needs roughly 70 seconds. The scheduling algorithm likely cannot maintain always the workers busy and consequently the value of 70 seconds is a lower bound of $R$. To simulate a high task arrival rate, as it is the case for high throughput computing, we must tune the period of time ($R$) between two scheduling event to generate a task arrival rate close as much as possible to the task completion rate without exceeding it. For the basic heuristics (MinMin, MaxMin and sufferage) and after a manual tuning, the best value we obtained for $R$ is 2 minutes.

To be able to evaluate the effectiveness of the different studied scheduling algorithms, we have used the same task arrival patterns. A task arrival pattern corresponds to a finite list of task arrivals generated over a period of two weeks. According to the above values, we generated a set of 50 task arrival patterns (50 trials). The same set of task arrival patterns is used for each simulated scheduling algorithm. These patterns are used to average each point in the following comparison charts. Vertical lines at the top of a bar indicate the minimum and the maximum values for the 50 trials. The makespan for each adapted heuristic has been normalized with respect to the non-adapted heuristic. Note that a unit value of the normalized makespan is specific to each month and is not comparable with another month.

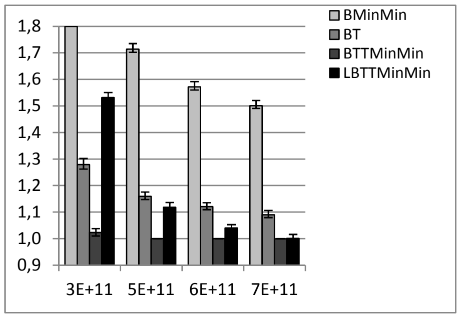
As we are not concerned in this paper with studying prediction techniques, the predicted average speed of a worker has been computed using the actual time series values. At a scheduling event time, an average worker speed was computed using the next time series values corresponding to the next fifteen minutes. All the computed average speeds are structured as times series. These average worker speed time series have been also used to determine the workers that belong to each $G^k_{s_n}$. The price associated with an average speed $s_n$ is $s_n$ divided by $s_0$, where $s_0$ is the minimum positive speed.

In the rest of this section, only the results obtained using MinMin heuristic will be presented. For the two other heuristics, namely MaxMin and Sufferage, we obtained comparable results as with MinMin. Using this heuristic, the simulated scheduling algorithms are :

1. the original version which is not adapted to the budget constraint and to the grid scale,
2. the adapted version to the budget limit without any optimization of the budget repartition as described by the figure 2, named BMinMin,
3. the version with a repartition of the budget among tasks according to the condition (4), named BTMinMin,
4. the version with a repartition of the budget among tasks and through time according to the condition (5), named BTTMinMin, and
5. the version adapted to a large number of workers with a repartition of the budget among tasks and through time, named LBTTMinMin.



(a) Using June time series



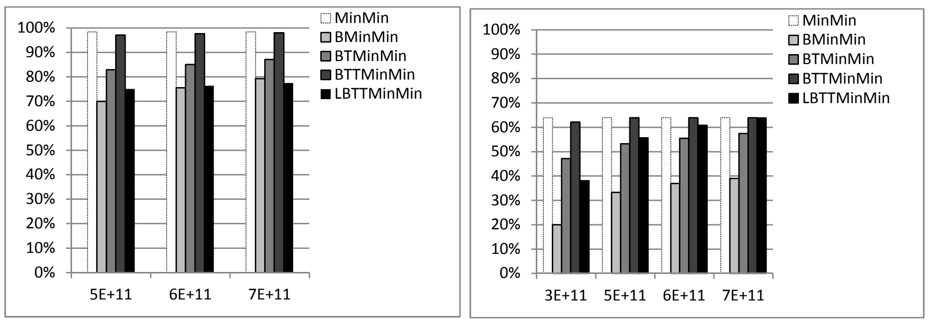
(b) Using September time series

**Fig. 4.** Normalized makespan of the adapted MinMin versions

Figure 4 shows the effect of varying the budget on the normalized makespan for the different adapted MinMin based algorithms. The conventional MinMin heuristic is not budget constrained, it gives the minimum makespan which, in average, corresponds to the two weeks task generation period plus ten minutes for September and plus 42 hours 30 minutes for June simulations. So for September, the MinMin heuristic achieves an execution rate close to the arriving rate, while for June an overload problem occurs. Indeed we had tuned the task arriving rate according to the ACS of September which is near the double of June. Nevertheless, for June simulations, even if the desktop grid is overloaded, the repartition of the budget among tasks and through time is able to enhance the makespan. It can also be observed that for all the MinMin budget adapted versions, increasing the budget decreases the makespan until it reaches a value close to that of the non constrained MinMin version.

For September series with a budget little more than $3*10^{11}$, the BTTMinMin version is also able to sustain the arrival rate whereas the other budget constrained versions (BMinMin, BTMinMin and LBTTMinMin) are not able to perform so much. The repartition of the budget among tasks, applied by the BTMinMin version, reduces significantly the normalized makespan. Using the BTTMinMin version, and so adding the repartition of the budget through time, improves more over the normalized makespan. Then, if we consider the LBTTMinMin version, which gathers workers into classes according to their speed and uses an approximated value of ready time for all workers in a same class, the normalized makespan suffers from this approximation as counterpart of the ability to support a large scale desktop grid. Notwithstanding, with a budget equal to $7*10^{11}$, the LBTTMinMin version is able to achieve a normalized makespan comparable to the BTTMinMin version. This is due to the fact that for this last budget the most part of the submitted computing tasks were executed by workers which are more expensive but have a larger availability interval. For such workers, which belong to a same class, the imprecision of the approximation of the ready time is quite small.

Figure 5 shows the average resource utilization rate of the original and the extended MinMin versions. As we have already noted, it can be seen that the desktop grid is overloaded in the June case: the average resource utilization is about 98% for the non-adapted MinMin version. Applying a budget constraint decreases the resource utilization. According to figures 4 and 5, we can observe that a lower makespan can be achieved by increasing the resource utilization. It is through an appropiate pricing that it would be possible avoiding the under utilization of resources.



(a) Using June time series　　(b) Using September time series

**Fig. 5.** Resource utilization rate of the original and adapted MinMin versions

## 6 Conclusion and future work

MinMin, MaxMin and Sufferage scheduling heuristics have been used in many grid execution environments and are becoming benchmarks. To reach high computing capacities and speeds, an economic incentive-based approach may be used. In this context, we have proposed a generic adapted algorithm to support independent task scheduling with a budget constraint onto large scale heterogeneous systems. The generic algorithm includes successive optimizations, to enhance the makespan, it computes a suited distribution of the budget among tasks and through time horizon. The simulation study, based on real traces, shows that the proposed optimizations are able to improve the makespan by a significant factor. For the scalability purpose, the computing resources are gathered according to their speed into classes; the algorithm maps the submitted tasks to these classes and then achieves a final scheduling for each worker class. The simulation results show that the adaptation to the grid scale does not degrade the makespan when the availability interval of computing node is not too small.

Our proposed algorithm rely on good prediction of worker speeds, we are now investigating this issue using time series prediction approach. Besides, the execution price at given speed should depend on the global offer and the global demand in the grid. For this purpose, a pricing algorithm is also studied in another ongoing work. To tackle malicious aborts, we are are looking to use a reputation mechanism to put aside the bad workers that try to degrade the performance of the scheduling system by intentionally accepting the execution of tasks and then aborting them.

## 7 Acknowledgements

We are grateful to XtremLab team for the use of XtremLab traces.

## References

1. D.P. Anderson, BOINC: A System for Public-Resource Computing and Storage, The $5^{th}$ IEEE/ACM International Workshop on Grid Computing, Pittsburgh, 2004, pp. 365-372.
2. D. Anderson, G. Fedak, The Computational and Storage Potential of Volunteer Computing, The $6^{th}$ Int'l Symposium on Cluster Computing and the Grid, Singapore, 2006, pp. 73-80.
3. The Berkeley Open Infrastructure for Network Computing, http://boinc.berkeley.edu/ [11 June 2010].
4. T. D. Braun, H. J. Siegel, N. Beck, A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems, J. Parallel and Distributed Computing, 61(6), 2001, pp. 107-131.
5. R. Buyya, D. Abramson, J. Giddy, Nimrod/G: An architecture for a resource management and scheduling system in a global computational Grid, The $4^{th}$ Int'l Conference and Exhibition on High Performance Computing in Asia-Pacific Region, IEEE Computer Society Press, Los Alamitos, CA, USA, 2000, pp. 283-289.
6. R. Buyya, D. Abramson, J. Giddy, Economy Driven Resource Management Architecture for Computational Power Grids, The Int'l Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA, 2000.

7. R. Buyya, J. Giddy, D. Abramson, An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications, The $2^{nd}$ Int'l Workshop on Active Middleware Services, Kluwer Academic Press, 2000.
8. R. Buyya, M. Murshed, D. Abramson, S. Venugopal, Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost-time optimization algorithm, Software: Practice and Experience, Vol. 35, No. 5, 2005; pp. 491-512.
9. R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the $5^{th}$ utility, J. Future Generation Computer Systems, 25(6), 2009, pp. 599-616.
10. H. Casanova, G. Obertelli, F. Berman and R. Wolski, The AppLeS parameter Sweep template: User-level middleware for the Grid, The Super Computing Conference, Dallas, Texas, 2000.
11. A. Chien, B. Calder, S. Elbert, K. Bhatia, Entropia: Architecture and performance of an enterprise desktop grid system, J. Parallel Distributed Computing, 63(5), 2001, pp. 597–610.
12. K. Cooper, A. Dasgupata, K. Kennedy, C. Koelbel, A. Mandal, G. Marin, M. Mazina, J. Mellor-Crummey, F. Berman, H. Casanova, A. Chien, H. Dail, X. Liu, A. Olugbile, O. Sievert, H. Xia, L. Johnsson, B. Liu, M. Patel, D. Reed, W. Deng, C. Mendes, Z. Shi, A. YarKhan, J. Dongarra, New Grid Scheduling and Rescheduling Methods in the GrADS Project, The Next Generation Software Workshop, Int'l Parallel and Distributed Processing Symposium, Santa Fe, IEEE CS Press, Los Alamitos, CA, USA, 2004.
13. H. Dail, H. Casanova, F. Berman, A Decoupled Scheduling Approach for the GrADS Environment, The IEEE/ACM Conference on Supercomputing, Baltimore, USA, IEEE CS Press, 2002.
14. G. Fedak, G. Germain, V. Neri, F. Cappello, XtremWeb: A Generic Global Computing System, The 1st IEEE/ACM Int'l Symposium on Cluster Computing and the Grid, Brisbane, Australia, 2001, pp. 582-587.
15. S. Fitzgerald, I. Foster, C. Kesselman, G. Von Laszewski, W. Smith, S. Tuecke, A Directory Service for Configuring High-Performance Distributed Computations, The $6^{th}$ IEEE Symposium on High-Performance Distributed Computing, Portland, OR, IEEE CS Press, Los Alamitos, 1997; pp. 365-375.
16. I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, J. Supercomputer Applications and High Performance Computing, 11(2), 1997, pp. 115–128.
17. R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith,T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel, Scheduling resources in multi-user, heterogeneous, computing environments with SmartNet, The $7^{th}$ IEEE Heterogeneous Computing Workshop, 1998, pp. 184-199.
18. . O. H. Ibarra, C. E. Kim, Heuristic algorithms for scheduling independent tasks on non-identical processors, J. ACM, 24(2), 1977, pp. 280-289.
19. D. Irwin, J. Chase, L. Grit, A. Yumerefendi, Self-Recharging Virtual Currency, $3^{rd}$ Workshop on Economics of Peer-to-Peer Systems SIGCOMM, Philadelphia, Pennsylvania, USA, 2005.
20. D. Kondo, G. Fedak, F. Cappello, A. Chien, H. Casanova, Resource Availability in Enterprise Desktop Grids, J. Future Generation Computer Systems, 23(7) 2006, pp. 888-903.

21. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, J. Parallel and Distributed Computing, 59(2), 1999, pp. 107-131.
22. S. Phatanapherom, P. Uthayopas, Unified Economic Deadline Scheduling Algorithm for Computational Grid, J. Information Technology, 11(4), 2005, pp. 13-22.
23. S. Venugopal, R. Buyya, A Deadline and Budget Constrained Scheduling Algorithm for eScience Applications on Data Grids, $6^{th}$ Int'l Conference on Algorithms and Architectures for Parallel Processing, Melbourne, Australia, 2005, pp. 60-72.
24. R. Wolski, N. Spring, J. Hayes, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, J. Future Generation Computing Systems, 15(5-6), 1999, pp. 757–768.
25. The XtremLab project, http://xw01.lri.fr:4320 [12 July 2007]
26. J. Yu, R. Buyya, Scheduling Scientific Workflow Applications with Deadline and Budget Constraints using Genetic Algorithms, J. Scientific Programming, 14(3-4), IOS Press, Amsterdam, The Netherlands, 2006, pp. 217-230.