Scheduling Moldable BSP Tasks[⋆]

Pierre-François Dutot
Laboratoire ID-IMAG
38330 Montbonnot, France

Marco A. S. Netto[†], Alfredo Goldman, Fabio Kon
Department of Computer Science
University of São Paulo, Brazil

Abstract

*Our main goal in this paper is to study the scheduling of parallel BSP tasks on clusters of computers. We focus our attention on special characteristics of BSP tasks, which can use fewer processors than the original required, but with a particular cost model. We discuss the problem of scheduling a batch of BSP tasks on a fixed number of computers. The objective is to minimize the completion time of the last task (*makespan*). We show that the problem is difficult and present approximation algorithms and heuristics. We finish the paper presenting the results of extensive simulations under different workloads.*

1 Introduction

With the growing popularity of Computational Grids [6] the model of environment in which parallel applications are executing is changing rapidly. In contrast to dedicated homogeneous clusters, where the number of processors and their characteristics are known *a priori*, Computational Grids are highly dynamic. In these new environments, the number of machines available for computation and their characteristics can change frequently. When we look at the case of Opportunistic Grid Computing, which uses the shared idle time of the existing computing infrastructure [8], the changes in machine availability occur even more rapidly. Thus, a model of parallel computation that does not allow variations in the number of processors available for computation would not fit well in this environment.

Moldable tasks are able to maximize the use of available resources in a dynamic Grid in the presence of fluctuations in machine availability. In this paper we extend the *Bulk Synchronous Parallel* (BSP) model [25] of computation to allow for the definition of moldable tasks that can be executed in a varying number of processors. As it will be described in detail later, a BSP application is a sequence of supersteps, composed of the execution of independent processes, separated by barrier synchronizations.

Due to complexity of the grid environment, we have first focused our work on the problem of scheduling tasks with a fixed set of available computers. However, we are currently investigating mechanisms to improve the scheduling by supporting preemption of BSP tasks so as to schedule malleable tasks. We are also studying mechanisms to propose a dynamic scheduling scheme. These improvements we allow us to develop sophisticated heuristics to schedule parallel applications on actual computational grids.

The remainder of this paper is organized as follows. At the end of this section we present our motivation and related work regarding the scheduling of moldable tasks. In Section 2 we describe the BSP model and we also discuss the moldability on BSP and the problem of scheduling moldable tasks. In Section 3 we propose an approximation algorithm and some heuristics providing complexity proofs. In Section 4 we show experimental results to evaluate the proposed algorithms. In Section 5, we close the paper with some final remarks and ideas for future works.

1.1 Motivation

Our group is developing a novel Grid middleware infrastructure called InteGrade [8]. The main principles of InteGrade are: modern object-oriented design, efficient communication based on CORBA, and native support for parallel computing. In the current version[*], the BSP model [9] for parallel computation is supported through an implementation of the BSPlib [12] library. In this paper, we propose new scheduling algorithms for batches of BSP tasks, which are being included into the InteGrade system.

Using only rigid BSP tasks, we could use classical results for scheduling tasks with different execution times and number of processors. However, in our grid environment we can easily reduce the number of processors of a BSP task, allocating two or more processes to the same processor. As our environment is based on CORBA, there are no

[⋆]Research supported by a grant from CNPq, Brazil - grant number: 55.2028/02-9.

[†]PhD student supported by a fellowship from CAPES, Brazil.

[*]Available for download at `http://gsd.ime.usp.br/integrade`

differences between local and remote communications, this is transparent to the programmer.

Given a BSP task that requires execution time t on n processors, we can allocate it without effort, depending on the memory constraints, using fewer processors. The behavior of moldability can be approximated by a discrete function. If fewer than n processors are available, say n', the execution time can be estimated by $t\lceil \frac{n}{n'} \rceil$.

1.2 Related work

Most existing works for scheduling moldable tasks are based on a two-phase approach introduced by Turek, Wolf, and Yu [24]. The basic idea is to select, in a first step, an allocation (the number of processors allocated to each task) and then solve the resulting non-moldable scheduling problem, which is a classical multiprocessor scheduling problem. As far as the makespan criterion is concerned, this problem is identical to a 2-dimensional strip-packing problem [1,4]. It is clear that applying an approximation of guarantee λ for the non-moldable problem on the allocation of an optimal solution provides the same guarantee λ for the moldable problem. Ludwig [14] improved the complexity of the allocation selection of the Turek's algorithm in the special case of monotonic tasks. Based on this result and on the 2-dimensional strip-packing algorithm of guarantee 2 proposed by Steinberg [21], he presented a 2-approximation algorithm for the moldable scheduling problem. These results however are designed for the general moldable tasks problem, where each task has a different execution time for each number of processors.

As we will see in the formal definition of BSP moldable tasks, the size of our instances is much smaller. This happens because we know the penalty incurred when the number of processors allocated to a task is different from the requested number of processors.

Mounié, Rapine and Trystram improved this 2-approximation result by concentrating more on the first phase (the allocation problem). More precisely, they proposed to select an allocation such that it is no longer needed to solve a general strip-packing instance, but a simpler one where better performance guarantees can be ensured. They published a $\sqrt{3}$-approximation algorithm [17] and later submitted a $3/2$-approximation algorithm [16, 18]. However, these results are for a special case of moldable tasks where the execution time decreases when the number of processors allocated to the task increases and the workload (defined as *time*$\times$*processors*) increases accordingly. We will see that this hypothesis is not verified here. To the best of our knowledge there is no other work on scheduling moldable BSP tasks.

2 The BSP Computing Model

The *Bulk Synchronous Parallel* model (BSP) [25] was introduced by Leslie Valiant as a bridging model, linking architecture and software. BSP offers both a powerful abstraction for computer architects and compiler writers and a concise model of parallel program execution, enabling accurate performance prediction for proactive application design.

A BSP abstract computer consists of a collection of virtual processors, each with local memory, connected by an interconnection network whose only properties of interest are the time to do a barrier synchronization and the rate at which continuous, randomly addressed data can be delivered. A BSP computation consists of a sequence of parallel supersteps, where each superstep is composed of computation and communication, followed by a barrier of synchronization.

The BSP model is compatible with conventional SPMD/MPMD (single/multiple program, multiple data), and is at least as flexible as MPI [15], having both remote memory (DRMA) and message-passing (BSMP) capabilities. The timing of communication operations, however, is different since the effects of BSP communication operations do not become effective until the next superstep.

The postponing of communications to the end of a superstep is the key idea for implementations of the BSP model. It removes the need to support non-barrier synchronizations among processes and guarantees that processes within a superstep are mutually independent. This makes BSP easier to implement on different architectures and makes BSP programs easier to write, to understand, and to analyze mathematically. For example, since the timing of BSP communications makes circular data dependencies among BSP processes impossible, there is no risk of deadlocks or livelocks in a BSP program. Also, the separation of the computation, communication, and synchronization phases allows one to compute time bounds and predict performance using relatively simple mathematical equations [20].

An advantage of BSP over other approaches to architecture-independent programming, such as the PVM [22] and MPI [10] message passing libraries, lies in the simplicity of its interface, as there are only 20 basic functions. A piece of software written for an ordinary sequential machine can be transformed into a parallel application with the addition of only a few instructions.

Another advantage is performance predictability. The performance of a BSP computer is analyzed by assuming that, in one time unit, an operation can be computed by a processor on the data available in local memory and based on the following parameters:

1. P – the number of processors;

2. w_i^s – the time to compute the superstep s on processor i;

3. h_i^s – the number of bytes sent or received by processor i on superstep s;

4. g – the ratio of communication throughput to processor throughput;

5. l – the time required to barrier synchronize all processors.

To avoid congestion, for every processor on each superstep, h_i^s must be no greater than $\lceil \frac{l}{g} \rceil$.

Moreover, there are plenty of algorithms developed for CGM (Coarse Grained Multicomputer Model) [5], which has the same principles of BSP, and can be easily ported to BSP.

Several implementations of the BSP model have been developed since the initial proposal by Valiant. They provide to the users full control over communication and synchronization in their applications. The mapping of virtual BSP processors to physical processors is hidden from the user, no matter what the real machine architecture is. BSP implementations developed in the past include: Oxford's BSPlib [12] (1993), JBSP [11] (1999), a Java version, PUB [2] (1999) and BSP-G [23] (2003).

2.1 Moldability on BSP

Given a BSP task that requires n processors, it is composed of n different processes which communicate on the global synchronization points. When designing BSP algorithms, for example using CGM techniques, one of the goals can be to distribute the load across processes more or less evenly.

To model moldability we use the following fact. When embedding BSP processes into homogeneous processors, if a single processor receives two tasks, intuitively, it will have twice as much work as the other processors. To reach each global synchronization, this processor will have to execute two processes and to send and receive the data corresponding to these processes. However, to continue processing, all the other processors have to wait. Hence, the program completion time on $n-1$ processors will be approximately two times the original expected time on n processors.

The same idea can be used when scheduling BSP tasks on fewer processors than the required. Each BSP process has to be scheduled to a processor and the expected completion time will be the original time multiplied by the maximum number of processes allocated to a processor. It is clear to observe that when processes are allocated to homogeneous processors, in order to minimize execution time the difference in the number of processes allocated to the most and to the least loaded processor should be at most one. This difference must be zero when the used number of processors divides the number of processes.

For the scheduling algorithms used in this paper, given a BSP task composed of n processes and with processing time t, if $n' < n$ processors are used, the processing time will be $t\lceil \frac{n}{n'} \rceil$. So, if only $n-1$ processors are available, the execution time of these tasks will be the same whether using $n-1$, or $\lceil \frac{n}{2} \rceil$ processors. Obviously, in the last case, we will have a smaller work area (number of processors times execution time).

2.2 Notations and properties

We are considering the problem of scheduling independent moldable BSP tasks on a cluster of m processors.

In the rest of the paper the number of processors requested by the BSP task i will be denoted req_i. The execution time of task i on a number p of processors will be $t_i(p)$. As we are dealing with BSP tasks, we can reduce the number of processors allocated to a task at the cost of a longer execution time. The relation between processor allocation and time is the following:

$$\forall q \forall p \in \left[\frac{req_i}{q+1}, \frac{req_i}{q} \right[,\ t_i(p) = (q+1)t_i(req_i)$$

where p and q are integers. In this work we do not consider a minimal number of processors for each task.

Table 1 shows an example with $req_i = 7$ and $t_i(req_i) = 1$, and the resulting workload which is defined as the product of processors allocated and execution times. We can see in this example that the workload is not monotonous in our case as in some other works on moldable tasks [17], but it is always larger than or equal to the workload with the required number of processors. Remark that for any task, on one processor the workload is equal to the minimum workload.

Table 1. A BSP task and its possible execution times and associated workloads.

#procs.	7	6	5	4	3	2	1
time	1	2	2	2	3	4	7
work	7	12	10	8	9	8	7

2.3 NP-hardness

The problem of scheduling independent moldable tasks is generally believed to be NP-hard, but this has never been formally proven. It contains as a special case the problem of scheduling independent sequential tasks (requiring only one processor), which is NP-hard [7]. However, the size of the moldable tasks problem is $O(n*m)$ since each task has to be defined with all its possible allocation, whereas the size of the sequential problem is $O(n + \ln(m))$ since we only need to know the number of available processors and the length of each task.

In the BSP moldable task problem, the problem size is hopefully much smaller, as we only need to know for each task the requested number of processors and the execution time for this required number of processors. The moldable behavior of the tasks is then deduced from the definition of BSP moldable tasks. Therefore the overall size of an instance is in $O(n*\ln(m))$ which is polynomial in both n and $\ln(m)$. The reduction from the multi-processor scheduling problem is then polynomial, which proves the NP-hardness of our problem.

3 Algorithms

To solve efficiently the problem of scheduling parallel BSP tasks, we have to design polynomial algorithm which provides on average a result close to the optimal. The first step is therefore to determine a good lower bound of the optimal value to be able to measure the performance of our algorithms. Two classic lower bounds for scheduling parallel tasks are the total workload divided by the number of available processors and the length of the longest task. With our previous notations, these two lower bounds are respectively $\sum_i t_i(req_i)/m$ and $\max_i t_i(req_i)$.

3.1 Guaranteed algorithm

The best way to assess the quality of an algorithm is to mathematically prove that for any instance, the ratio between the makespan ω of the schedule produced by the algorithm and the optimal makespan ω^* is bounded by a constant factor ρ. As we said in the introduction, the problem of scheduling independent moldable tasks has already been studied and some guaranteed algorithms have already been proposed for this problem. The best algorithm to date is a $3/2$ approximation algorithm proposed by Mounié et al. [18], however this algorithm needs an additional monotonicity property for the tasks. This property states that the workload is non decreasing when the number of processors allocated to a task increases which is clearly not the case with our moldable BSP tasks. An older algorithm which does not require this monotonic property has been designed by Ludwig [14]. This algorithm has a performance ratio of 2 as does the one we are proposing below, however it is much more complicated to use since it involves a strip packing phase. This is why we decided to design a 2-approximation algorithm based on our knowledge of the BSP tasks.

The algorithm is based on the dual approximation scheme as defined by [13]. The dual approximation scheme is based on successive guess $\hat{\omega}$ of the optimal makespan, and for each guess runs a simple scheduler which either outputs a schedule of makespan lower or equal to $2\hat{\omega}$, or outputs that $\hat{\omega}$ is lower than the optimal. With this scheduler and a binary search, the value of $\hat{\omega}$ quickly converges toward a lower bound of the optimal makespan for which we can produce a schedule in no more than $2\hat{\omega}$ units of time.

The scheduler works as follows. Based on the guess $\hat{\omega}$, we determine for each task i the minimal allocation a_i (if it exists) such that $t_i(a_i) \leq 2\hat{\omega}$. If there is a task such that this a_i does not exists (i.e. $t_i(req_i) > 2\hat{\omega}$) the optimal makespan is larger than this particular $t_i(req_i)$ and therefore larger than $\hat{\omega}$. Given these a_i, we schedule all the tasks that require more than one processor ("large" tasks) on exactly a_i processors, and we schedule the remaining tasks ("small" tasks = requiring exactly one processor) on the q remaining processors with a largest processing time first order.

There are three cases in which this algorithm fails to produce a schedule in no more than $2\hat{\omega}$ units of time:

1. There are too many processors required by "large" tasks ($\sum_{a_i>1} a_i > m$).
2. There are no processors left for "small" tasks ($\sum_{a_i>1} a_i = m$ and $\sum_{a_i=1} a_i > 0$).
3. One of the sequential tasks is scheduled to complete after the $2\hat{\omega}$ deadline. As the first fit has a 2-approximation ratio, it means that there is too much workload for "small" tasks ($\sum_{a_i=1} t_i(1) > (m - \sum_{a_i>1} a_i)\hat{\omega}$).

For each case we will prove that if the schedule fails, the guess $\hat{\omega}$ is lower than the optimal makespan. Before going into details for each case, we need to prove the following lemma:

Lemma 1 *For all task i such that $a_i > 1$, we have $t_i(req_i)req_i \geq a_i\hat{\omega}$.*

The idea behind this lemma is that the a_i processors allocated to task i are used efficiently for a sufficient period of time.

Proof. For a_i equal to 2, we know that $t_i(a_i - 1) > 2\hat{\omega}$ as a_i is the minimal number of processors to have an execution time no more than $2\hat{\omega}$. As we noted in Section 2.2 the workload on one processor is equal to the minimal workload $req_i t_i(req_i)$, therefore we can write when $a_i = 2$ and $t_i(a_i - 1) = req_i t_i(req_i)$ that $t_i(req_i)req_i \geq a_i\hat{\omega}$.

For the other extremal case, when $a_i = req_i$, since $req_i \geq 2$ we have $req_i - 1 \geq req_i/2$ and then $t_i(req_i - 1) =$

$2t_i(req_i)$ by definition of the execution times (see Section 2.2). By definition of a_i, we then have $2t_i(req_i) > 2\hat{\omega}$ and then $req_i t_i(req_i) > a_i\hat{\omega}$.

For the general case where $2 < a_i < req_i$, by definition of $t_i(a_i)$, there exists an integer q such that $t_i(a_i) = (q+1)t_i(req_i)$. As a_i is minimum, $t_i(a_i-1) > 2\hat{\omega}$ and there exists also an integer $s \geq 1$ such that $t_i(a_i-1) = (q+s+1)t_i(req_i)$. Therefore we have the following lower bound for $t_i(req_i)$:

$$t_i(req_i) > \frac{2\hat{\omega}}{q+s+1} \quad (1)$$

By definition of the execution times, as $t_i(a_i-1) = (q+s+1)t_i(req_i)$, we have $a_i - 1 < req_i/(q+s)$ which can be rewritten as:

$$req_i \geq (q+s)(a_i-1)+1 \quad (2)$$

By combining inequalities 1 and 2, we have a lower bound for the left term of the lemma:

$$t_i(req_i)req_i > \frac{2((q+s)(a_i-1)+1)}{q+s+1}\hat{\omega} \quad (3)$$

In order to conclude, we have to compare the values of a_i and $2((q+s)(a_i-1)+1)/(q+s+1)$ which is done by comparing their difference:

$$\begin{aligned} 2((q+s)(a_i-1)+1) - a_i(q+s+1) &= \\ 2qa_i + 2sa_i - 2q - 2s + 2 - qa_i - sa_i - a_i &= \\ q(a_i-2) + s(a_i-2) - (a_i-2) &= \\ (q+s-1)(a_i-2) & \end{aligned}$$

This value being positive or equal to zero, $a_i\hat{\omega}$ is a lower bound of the right term of inequality 3, which concludes the proof of the lemma. □

Theorem 1 *When the schedule fails, the guess $\hat{\omega}$ is too small.*

Proof.

Case 1

$$\sum_{a_i>1} a_i > m$$

In this case the minimal total workload $\sum_i req_i t_i(req_i)$ can be bounded in the following way:

$$\begin{aligned} \sum_i req_i t_i(req_i) &\geq \sum_{a_i>1} req_i t_i(req_i) \\ &\geq \sum_{a_i>1} a_i\hat{\omega} \\ \sum_{a_i>1} a_i\hat{\omega} &> m\hat{\omega} \end{aligned}$$

Therefore $\hat{\omega}$ is lower than the optimal makespan.

Case 2

$$\sum_{a_i>1} a_i = m \text{ and } \sum_{a_i=1} a_i > 0$$

As previously, we can bound the minimal total workload but this time the strong inequality is the first one:

$$\begin{aligned} \sum_i req_i t_i(req_i) &> \sum_{a_i>1} req_i t_i(req_i) \\ \sum_{a_i>1} req_i t_i(req_i) &\geq \sum_{a_i>1} a_i\hat{\omega} \\ \sum_{a_i>1} a_i\hat{\omega} &= m\hat{\omega} \end{aligned}$$

Which again proves that the guess was too small.

Case 3

$$\sum_{a_i=1} t_i(1) > \left(m - \sum_{a_i>1} a_i\right)\hat{\omega}$$

Finally in this case, the bounding is a little more subtle:

$$\begin{aligned} \sum_i req_i t_i(req_i) &= \sum_{a_i>1} req_i t_i(req_i) + \sum_{a_i=1} req_i t_i(req_i) \\ &\geq \sum_{a_i>1} a_i\hat{\omega} + \sum_{a_i=1} t_i(1) \\ &> \sum_{a_i>1} a_i\hat{\omega} + \left(m - \sum_{a_i>1} a_i\right)\hat{\omega} = m\hat{\omega} \end{aligned}$$

Therefore in all the cases where the schedule fails, the guess was lower than the optimal makespan. □

Corollary 1 *The proposed algorithm provides a 2-approximation for BSP moldable tasks.*

The sum of the sequential execution times of all the tasks is an upper bound of the optimal makespan, which is polynomial in the size of the instance. Starting from this guess, we can use the algorithm in a binary search of the lowest possible value $\hat{\omega}$ for which we can build a schedule in at most $2\hat{\omega}$. If $\epsilon/2$ is the size of the last step of the binary search, $\hat{\omega} - \epsilon/2$ is a lower bound of the optimal ω^*, and $2\hat{\omega} < 2\omega^* + \epsilon$ which means that the schedule produced in the last step is at most $2 + \epsilon$ times longer than the optimal.

3.2 Tested heuristics

We have implemented four algorithms to schedule a set of BSP tasks, each task comprising a set of processes, on homogeneous processors.

The **first algorithm** $A1$ is the well-known Largest Task First list scheduling (where largest refers to *number of processors*$\times$*execution time* i.e. the workload) with a pre-processing stage. This pre-processing consists of modifying all tasks regarding the maximum number of processors $maxnprocs$ each one will receive. The idea here is to reduce the size of the largest jobs in order to have less heterogeneity in the set of tasks.

When the original number of processors $reqnprocs$ of a task is modified, the amount of time $reqtime$ needed to execute it is also modified. The pseudo-code below is executed on each task before scheduling.

Algorithm 1 Pseudo-code to pre-processing each task to be scheduled in algorithm $A1$.

```
if $task.reqnprocs > maxnprocs$ then
    $task.reqtime = \lceil (task.reqnprocs/maxnprocs) \rceil * task.reqtime$
    $task.reqnprocs = maxnprocs$
end if
```

The main problem of this algorithm is that we must verify all possible $maxnprocs$ values, from one to the number of processors available in the computing system so as to discover the most appropriated value. Doing this we noticed that the true LTF scheduling (i.e. when $maxnprocs = m$ tasks are not reduced) was usually far from the optimal makespan.

Once the tasks are reduced they are sorted according to their sizes in $O(n * \ln(n))$ steps, and then scheduled in n steps. The overall complexity of this algorithm is therefore $O(m * n * \ln(n))$.

The **second algorithm** $A2$ is based on the idea of reducing the idle time in the schedule by optimizing the placement of the different tasks (see Figure 1). The algorithm comprises two steps:

1. Look for the *best* task such that, when scheduled, the idle time is reduced or remains the same. *Best* task means the smallest amount of idle time, the better the task. Note that in this step, the number of processors and time to execute the task can be modified. If a task is found, schedule it.
2. If Step 1 has failed, schedule the first largest task that was not scheduled yet.

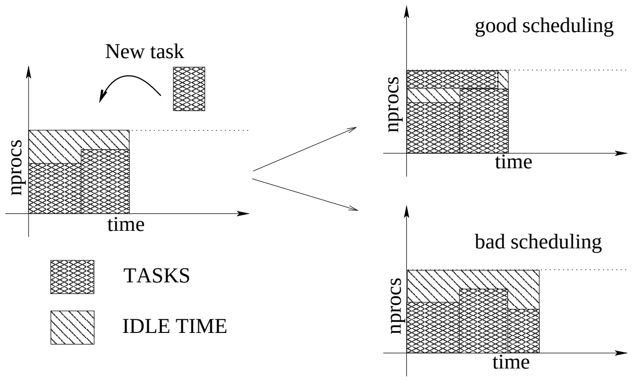


Figure 1. Examples of schedulings to reduce the idle time.

As we have seen in the presentation of the BSP moldable model, for a given task there can be several allocations having the same execution time. For example, in Table 1 the allocations to 4, 5 and 6 processors all have an execution time of 2. We therefore will only consider here interesting allocations, for which there is no smaller allocation for the same execution time.

With this restriction the number of possible allocations goes down from $reqnprocs$ to approximately $2\sqrt{reqnprocs}$. This greatly reduces the complexity of the algorithm, however the overall complexity is still greater than $O(n * \ln(m))$ which is the size of the instance.

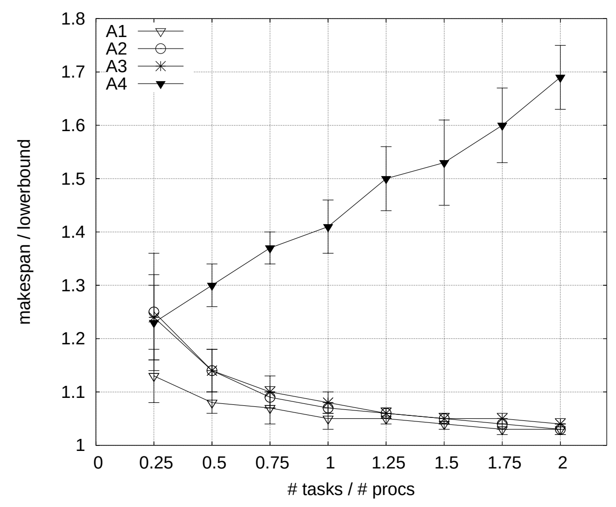
The **third algorithm** $A3$ is a derivation of the second one previously presented. It basically consists of scheduling tasks that generate the smallest idle time, even if the new idle time is greater than the original one. Thus, the first step presented in the previous algorithm is not limited to smaller idle times, and the second step is never executed.

The **fourth algorithm** $A4$ is the guaranteed algorithm presented in the previous section. It is the fastest algorithm, however we will see that its average behavior is far from the best solutions found.

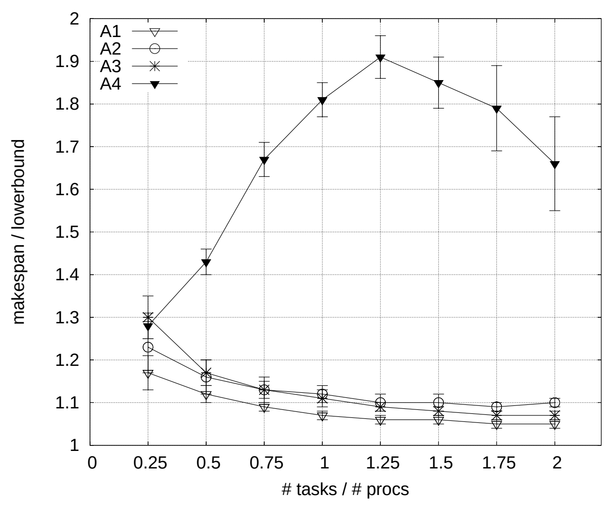
4 Experimental Results

In order to evaluate the algorithms, we developed a simulator that implements the presented algorithms and used both real and generated workloads. The real workloads[§]

[§]Available at: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

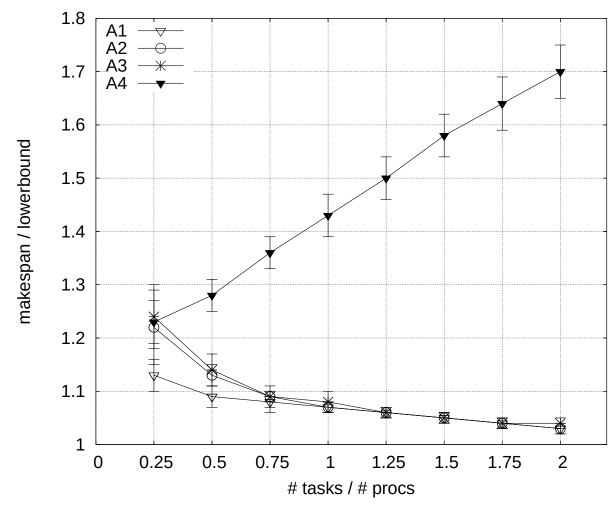


(a) Real Workloads

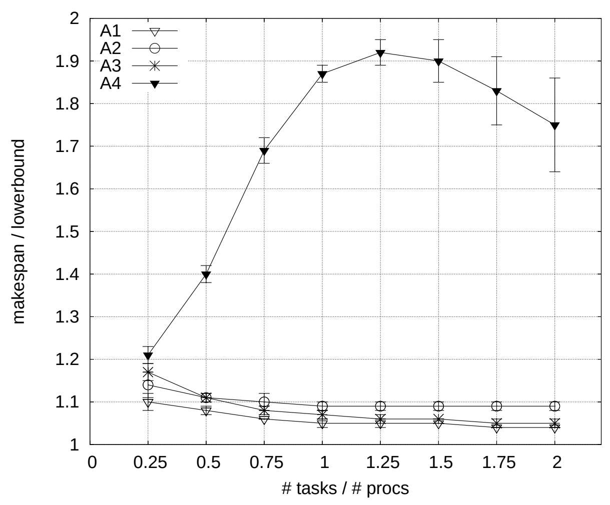


(b) Generated Workloads

Figure 2. Evaluation of the scheduling algorithms on 64 processors.



(a) Real Workloads



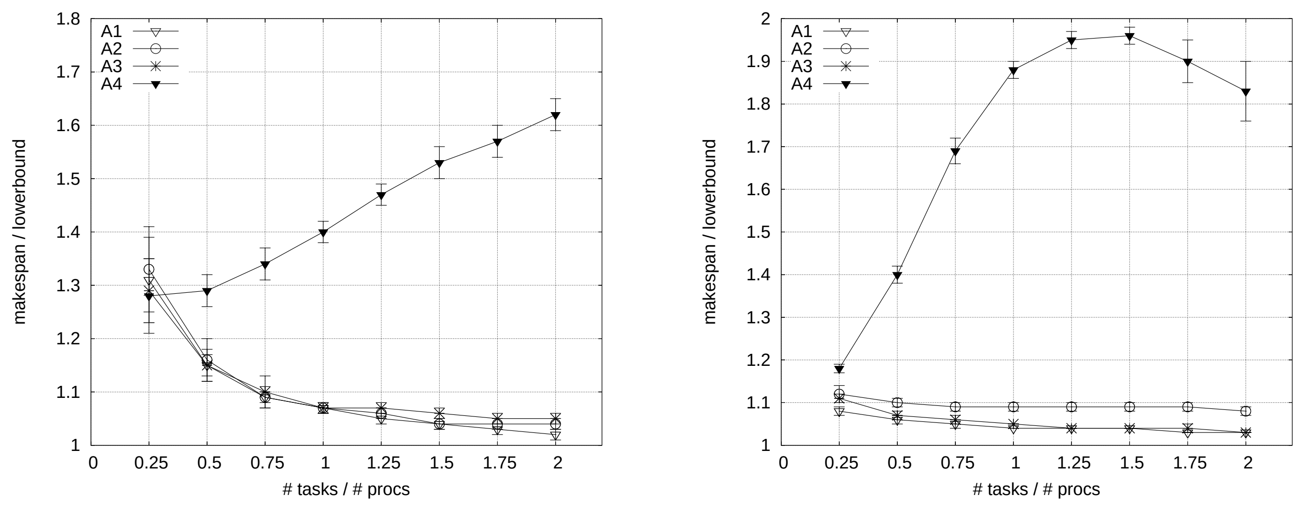
(b) Generated Workloads

Figure 3. Evaluation of the scheduling algorithms on 128 processors.

are from two IBM SP2 systems located at Cornell Theory Center (CTC) and San Diego Supercomputer Center (SDSC) [19], and the generated workloads were generated by a Gaussian distribution. Unlike the real workloads, the number of processors requested by the tasks in the generated instances are in most cases not powers of two [3]. Note that although the real workloads are not from execution of parallel BSP tasks, the selected machines work with regular parallel applications, and to the best of our knowledge there should be no difference between workloads of MPI and BSP applications.

To perform the experiments we chose three different platforms: with respectively 64, 128 and 256 processors. We selected the SDSC workloads to evaluate the algorithms on 64 and 128 processors and the CTC workloads were used in the experiments with 256 processors. The generated workloads were used for all platforms.

For each experiment we performed 40 executions with different workloads, and then we took out the five best and the five worst results to reduce the deviation. The tasks in each real workload experiment were selected randomly from all the tasks in the corresponding logs. The graphics illustrated in Figures 2, 3 and 4 depict the results obtained in our experiments. In these figures the x-axis is the ratio between the number of tasks scheduled and the number of processors of the computer, while the y-axis is the ra-



(a) Real Workloads (b) Generated Workloads

Figure 4. Evaluation of the scheduling algorithms on 256 processors.

tio between the schedule length and a lower bound of the optimal makespan for the considered instance. This lower bound is actually the maximum of the two classical lower bounds: the execution time of the longest task (when allocated to its required number of processors) and the minimal average workload per processor. The schedule produced by the fourth algorithm is always lower or equal to two times the average workload, as can be deduced from the proof of Theorem 1.

Based on the results we can observe that algorithm $A1$ generally produces the best schedules. The algorithms $A2$ and $A3$ have similar behaviors and are very close to $A1$. Finally, as expected the fourth algorithm has a ratio which is close to 2 in the unfavorable cases. Remark that for the generated workload, the worst results of $A4$ are for tasks/processors ratios close to 1. This result confirms the intuition [16] that for moldable task problems the difficult part is when there are approximately as many tasks as processors.

To illustrate the difference between the fourth algorithm and the three other algorithms, we included Figure 5 and 6 that depict two schedules for 20 tasks on 16 processors respectively made with the third and the fourth algorithm. On Figure 6 it appears clearly that reducing all the tasks to the allocation which is the smallest below the $2\hat{\omega}$ limit tends to produce schedules close to twice the optimal, since most of the tasks are sequential.

As mentioned previously, the main problem of the algorithm $A1$ is that we need to schedule the tasks several times in order to discover the threshold, which is the maximum amount of processors the tasks should use. However, when there is a small number of processors in the computing environment, this algorithm is still usable in reasonable time.

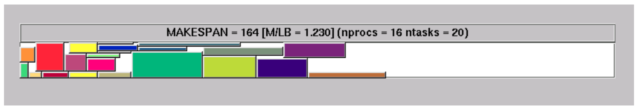


Figure 5. A schedule of 20 tasks on 16 processors with algorithm A3.

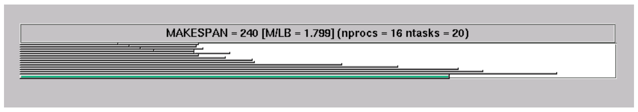


Figure 6. A schedule of 20 tasks on 16 processors with algorithm A4.

For larger numbers of processors, the algorithms $A2$ and $A3$ should be used, since even if they do not produce the best results, the difference is within reasonable bounds. As we could have guessed, the longer it takes to schedule the tasks, the better the results.

This is illustrated in Figure 7, where the execution times of the four algorithms are compared on 64 processors for 10 to 100 tasks. As previously described, the fourth algorithm is much faster than the three others, and the slowest algorithm is the first one. The execution times on the time scale are in milliseconds. For larger instances (1024 tasks on 512 processors) we witnessed execution times of several minutes on a recent computer (Pentium III 800 MHz, 512MB RAM). We ran all the other experiments on the same computer.

Another important observation is that the results using

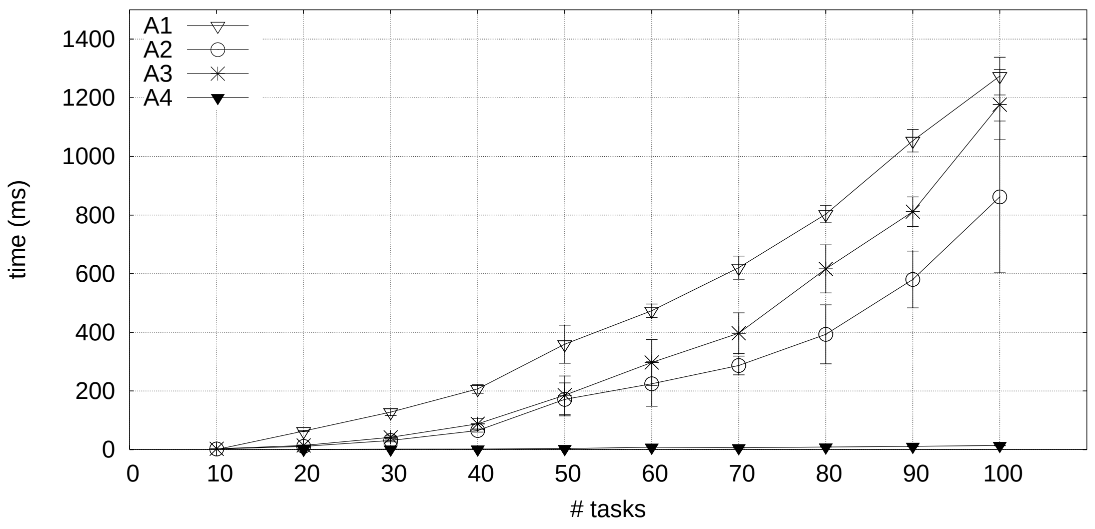


Figure 7. Execution times for up to 100 tasks on 64 processors.

real and generated workloads are similar for the algorithms $A1$, $A2$ and $A3$. Our main goal to make experiments with generated workloads is that the real workloads are mostly made of regulars tasks, as well as tasks requiring processors in powers of two. These characteristics are usually found only in dedicated computer systems, such as supercomputers and clusters. Thus, we have used workloads with other characteristics in order to verify the quality of the proposed algorithms on different environments.

5 Conclusion and Future Work

In this paper we studied the scheduling of moldable BSP parallel tasks. First we showed that the problem is NP-hard, and then we provided a 2-approximation algorithm and some good heuristics. On the algorithms, the number of processors given to a task with n processes can range from 1 to n. However, due mainly to memory limitations this may not be feasible in practice. Moreover, with few processors the task can be delayed for long time. Thus, as future work we intend to limit the minimal number of processors for a task in order to limit the maximal number of processes in each processor.

This work has as its final goal an implementation to be used to schedule parallel applications on our grid environment, InteGrade. Also as future works we intend to explore the possibilities provided by our grid environment, processors heterogeneity, parallel tasks preemption, and machine unavailability. For the last two cases we will study in detail the effects of interrupting a parallel task and possibly continue to execute it on a different number of processors, which is possible with the BSP synchronizations and our already implemented checkpointing library.

References

[1] R. Baker, E. G. Coffman, and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The paderborn university bsp (pub) library. *Parallel Computing*, 29(2):187–207, 2003.

[3] W. Cirne and F. Berman. A model for moldable supercomputer jobs. *In Proceedings of the 15th International Parallel & Distributed Processing Symposium*, April 2001.

[4] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[5] F. Dehne. Coarse grained parallel algorithms. *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, 24(3–4):173–176, 1999.

[6] I. Foster and C. Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, 2003.

[7] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W. H. Freeman, New York, 1979.

[8] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16:449–459, March 2004.

[9] A. Goldchleger, C. A. Queiroz, F. Kon, and A. Goldman. Running highly-coupled parallel applications in a computa-

tional grid. *In Proceedings of the 22th Brazilian Symposium on Computer Networks*, 2004.

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[11] Y. Gu, B.-S. Lee, and W. Cai. JBSP: A BSP Programming Library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.

[12] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[13] 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.

[14] W. T. Ludwig. *Algorithms for scheduling malleable and nonmalleable parallel tasks*. PhD thesis, University of Wisconsin - Madison, Department of Computer Sciences, 1995.

[15] Message Passing Interface Forum. MPI: A Message Passing Interface. *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[16] G. Mounié. *Efficient scheduling of parallel application : the monotonic malleable tasks*. PhD thesis, Institut National Polytechnique de Grenoble, June 2000. Available in french only.

[17] G. Mounie, C. Rapine, and D. Trystram. Efficient approximation algorithm for scheduling malleable tasks. *In Proceedings of the 11th ACM Symposium of Parallel Algorithms and Architecture*, pages 23–32, 1999.

[18] G. Mounie, C. Rapine, and D. Trystram. A $\frac{3}{2}$-approximation algorithm for independent scheduling malleable tasks. Submitted for publication 2001.

[19] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions Parallel & Distributed Systems*, 12(6):529–543, June 2001.

[20] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Journal of Scientific Programming*, 6:249–274, 1997.

[21] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM Journal on Computing*, 26(2):401–409, 1997.

[22] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[23] W. Tong, J. Ding, and L. Cai. A parallel programming environment on grid. *In Proceedings of the International Conference on Computational Science*, volume 2657 of *Lecture Notes in Computer Science*, pages 225–234. Springer, 2003.

[24] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms for scheduling parallelizable tasks. *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 323–332, San Diego, California, June 29–July 1, 1992. SIGACT/SIGARCH.

[25] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 1990.