Scheduling Moldable BSP Tasks on Clouds

Thiago Kenji Okada[1], Marcos Amarís González[1], Alfredo Goldman vel Lejbman[1]

[1]Instituto de Matemática e Estatística (IME)
Universidade de São Paulo (USP)
R. do Matão, 1010 – Vila Universitária, São Paulo – SP, 05508-090

Abstract. *The focus of this study is on the scheduling of moldable Bulk Synchronous Parallel (BSP) tasks on cloud computing environments. "Moldable" in this context is related to possibly reducing the number of required processors of a BSP task, recalculating the total execution time using a particular cost model. From there, we analyze how a moldable BSP task is influenced by the unreliable behavior of clouds, simulating the execution of several BSP tasks in existing public cloud computing environment. The objective of this paper is to analyze the difference between the completion time of the last task (*makespan*), in both a simulated setting and in real clouds, analyzing the results and concluding how much a theoretical model can be useful in such environments.*

Resumo. *O foco deste estudo é escalonamento de tarefas BSP moldáveis em ambientes de computação em nuvem. "Moldável" neste contexto está relacionado em possivelmente reduzir o número de processadores requisitados por uma tarefa BSP, recalculando seu custo de execução total a partir de um modelo de custo. A partir daí, analisamos como uma tarefa BSP moldável é influenciada pelo comportamento imprevisível das nuvens, simulando a execução de diversas tarefas BSP em nuvens computacionais públicas. O objetivo desse artigo é analisar a diferença entre o tempo de término da última tarefa (*makespan*), tanto em simulação quanto em nuvens reais, analisando os resultados e concluindo o quanto um modelo teórico é próximo a realidade.*

1. Introduction

With the advent of cloud computing, it is not necessary anymore to invest large amounts of money on computing resources. Instead, it is possible to obtain processing or storing resources, and even complete systems, on demand, using one of the several available services from cloud providers like Amazon EC2, Microsoft Azure and Google Compute Engine.

Therefore, both the IT and academic community started to see the benefits that cloud computing brings, like resources flexibility, live migrations and reduced costs both on hardware setup and on the maintenance of the system [Armbrust et al. 2010, Mell and Grance 2011]. However, problems like low networking performance, performance variation during rush hours and additional computing overhead are the new challenges that cloud computing brings to High Performance Computing (HPC) applications [Wei and Blake 2010, Yuvaraj and Palanivel , Chandio et al. 2014, Nuaimi et al. 2012].

Efficient parallel algorithms in HPC, based on Coarse Grained Multicomputer (CGM) model, seek to evenly distribute the workload between the available resources and to minimize the number of communication rounds [Dehne et al. 1999]. These kind of algorithms can be implemented directly on the Bulk Synchronous Parallel (BSP) model [Valiant 1990]. So, these algorithms can both manage resources better and reduce the total computing time [Calinescu 1997].

In this context, the objective of this work is to study the scheduling of BSP tasks in cloud computing environments. We simulate tasks coming from real HPC traces, schedule them using different algorithms presented in the literature and analyze the impact of public clouds in these tasks, simulating the execution in both Microsoft Azure and Google Compute Engine.

The remainder of this paper is structured as follows: In Section 2, we review the literature about the area. In Section 3, we introduce the BSP model, and describe the idea of moldable BSP tasks. In Section 4, we describe the scheduling algorithms presented in the literature implemented in this work. In Section 5, we describe our experiments and methodology. In Section 6, we present the results from the experiments. Finally, in Section 7, we present the conclusion of this work and future work.

2. Related Work

In [Dutot et al. 2005] the authors analyze the problem of scheduling moldable BSP tasks in computational grids, demonstrating that the scheduling of BSP tasks is a NP-hard problem. Afterwards, they propose four different algorithms to schedule BSP tasks in polynomial time. However, in this paper all experiments are done in a computational grid simulator, where the results are deterministic by nature. In our work we benchmark the algorithms described in this paper in real public clouds, where the performance is not deterministic.

In [Hunold 2015] the author developed a new algorithm to schedule moldable tasks with precedence constraints, and for arbitrary speedup functions. It shows the importance of new studies with moldable tasks for HPC applications, and while they do all experiments in a simulation, they comment the importance of real world testing.

In [Huang et al. 2013, Huang et al. 2014], the authors proposes an algorithm to schedule moldable tasks using a similar concept to SaaS (*Software as a Service*). They call this approach HPCaaS (*HPC as a Service*). In this paper, they show the importance of moldable tasks in modern HPC computing, thanks to the popularity of *Message Passing Interface* (MPI) and clouds in HPC applications.

In [Gonçalves et al. 2015] the authors developed a strategy to predict performance of applications running on clouds based on the observed performance for certain resources configurations. In this paper the authors had problems with their accuracy in some tests thanks to the fluctuations on cloud performance, showing how unreliable cloud computing performance is sometimes.

Our work serves as a bridge between the literature and real world, by testing scheduling algorithms presented in the literature in existing public clouds and comparing the makespan between real clouds and simulation environments.

3. The BSP model

Parallel computing models, like PRAM, LogP and others, have been an active research topic since the development of modern computers [Gibbons et al. 1998, Juurlink and Wijshoff 1998, Skillicorn and Talia 1998]. Their main goal is to provide a standard way of describing and evaluating the performance of parallel applications. For the success of a parallel computing model, it is paramount to also consider the characteristics of the underlying architecture of the hardware being used.

One of the most well-established models for parallel computing is the Bulk Synchronous Parallel (BSP), first introduced by Valiant in 1990 [Valiant 1990]. Its main goal was to provide a bridging model that can represent different architectures sufficiently well, without considering all the hardware details. The BSP model bridges the essential characteristics of different kinds of machines as a combination of three attributes:

- a set of virtual processors, each associated to a local memory;
- a router, that delivers the messages in a point-to-point manner;
- a synchronization mechanism for all or for a subset of processors.

The computation is organized in a sequence of *supersteps*, each one divided into three successive—logically disjointed—phases. On the first phase, all processors use their local data to perform local sequential computations in parallel (i.e., there is no communication among the processors.) The second phase is a communication phase, where all nodes exchange data performing personalized all-to-all communication. The last phase consists of a global synchronization barrier, that guarantees that all messages were delivered and all processors are ready to start the next superstep. Figure 1 depicts the phases of a BSP application. On the BSP model, there is no restriction on when to send the messages, but all of them should be received by the synchronization barrier. According to the execution model, the first and second phase may occur simultaneously.

The BSP model has been widely used on different applications contexts. HPC practitioners have been using the BSP model to design algorithms and software that can run on any standard architecture with guaranteed performance [Kirtzic 2012]. Modelling a task using the BSP model is useful to preview the performance of algorithms, and facilitates its implementation.

Consider a BSP program that runs on S supersteps using p processors simultaneously with clock rate (speed) R. Let g (the *gap*) be the bandwidth of the network and L the latency—i.e., the minimum duration of a superstep—which reflects not only the latency of the network, but also the overhead of the synchronization step.

The cost to execute the i-th superstep is then given by:

$$w_i + gh_i + L \tag{1}$$

where w_i is the maximum amount of local computations executed, and h_i is the largest number of packets sent or received by any processor during the superstep. If $W = \sum_{i=1}^{S} w_i$ is the sum of the maximum work executed on all supersteps and $H = \sum_{i=1}^{S} h_i$ the sum of the maximum number of messages exchanged in each superstep, then the total execution time of the application is given by:

$$T = W + gH + LS \tag{2}$$

It is common to present the parameters of the BSP model as a tuple (w, g, h, L).

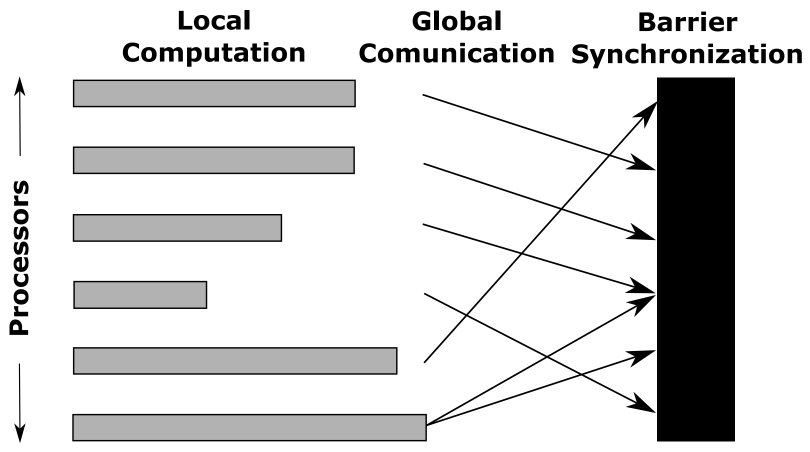

Figure 1. Superstep in a Bulk Synchronous Parallel Model.

There are other parametrized parallel models [Juurlink and Wijshoff 1998, Skillicorn and Talia 1998], almost all of them using or extending the core of the BSP model. In [Dehne et al. 1993], the author has studied the problem of designing scalable parallel geometric algorithms, and he has introduced the Coarse Grained Multicomputer model (CGM). In this model a set of p processors each with an $O(N/p)$ local memory, where N is the input size of the problem. There are plenty of algorithms developed for CGM, and these algorithms can easily be ported to BSP [Cáceres et al. 1997].

3.1. Moldability of BSP tasks

The idea of a moldable BSP task is to possibly reduce the number of processors from a task, allowing it to run with less resources than originally planned. Of course, this would impact the total run time of the task, but thanks to the BSP model we can preview its new run time.

We will model the time of a new task using the following fact. Assuming each task uses the same time, and a task needs n processors. If we have $n - 1$ processors instead, we will need to allocate one of the tasks in another processor. If a processor receives two processing tasks instead of one, it's easy to realize that the task as a whole needs twice as much time to be concluded.

This occurs because even if the remaining processors already concluded their processing task, they need to wait for the processor with the additional tasks to conclude its work to proceed to the next superstep [Cordeiro et al. 2013]. Therefore in the BSP model the execution time for $n - 1$ or $\frac{n}{2}$ is approximately the same, about twice as much than the time to execute the task with n processors [Dutot et al. 2005].

Generalizing the idea, if we have less than the number of requested processors by the task, we may redistribute the tasks evenly between the number of available processors. Assuming each task have the same run time, we conclude that the new task run time is:

$$t' = t\lceil \frac{n}{n'} \rceil \tag{3}$$

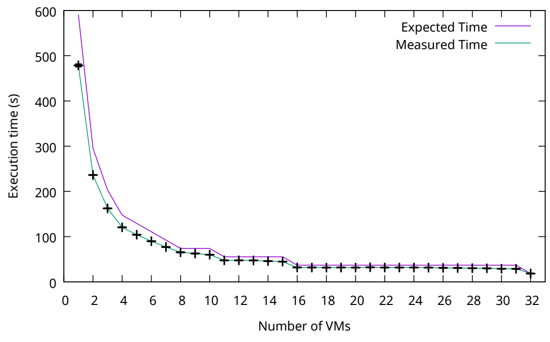

Figure 2. Microsoft Azure

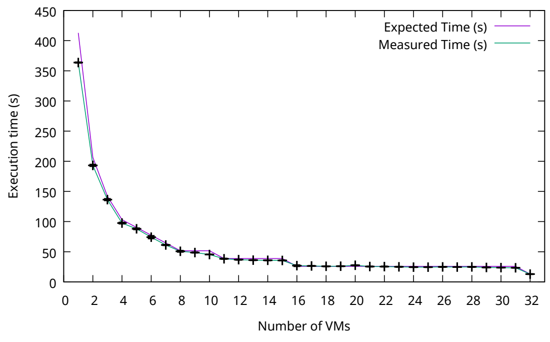

Figure 3. Google Compute Engine

Where t is the original task run time and t' is the new task run time.

To test this in cloud environments, we did an experiment using 32 moldable tasks (1024x1024 matrix multiplication), running different number of VMs with one processor, in both Google Compute Engine (using `n1-standard-1` instance) and Microsoft Azure (using `Standard A1` instance), distributing the 32 moldable tasks evenly between the available VMs. Each test configuration was repeated 10 times.

Figure 2 shows a plot of the data for Microsoft Azure, while Figure 3 is the same for Google Compute Engine. *Expected* is the time run time calculated from 32 VMs using Formula 3, while *Measured* is the measured time after 10 experiments.

This experiment shows that even if not perfect, calculating the new execution time by the formula described above is a pretty accurate way to preview the new execution time of a moldable task. So, in this paper we will assume that the processing time of each BSP task will be $t\lceil \frac{n}{n'} \rceil$, as explained above.

4. Scheduling BSP tasks in cloud

If scheduling of BSP tasks was an easy problem, this work wouldn't be interesting. However, in [Dutot et al. 2005], they demonstrate that the problem of scheduling moldable BSP tasks is a polynomial time reduction from the multi-processor scheduling problem, proving that our problem is NP-hard. So instead of trying to solve the problem in an optimal way, we will use scheduling algorithms based on heuristics found in the literature.

Because of limitations in cloud computing environments, we need to schedule the tasks in two stages: we first schedule the tasks considering only one computer, after that we distribute the tasks between the available VMs. This second stage is necessary because in clouds you can have different combinations of virtual machines (VM) that may result in the same number of resources. For example, if you need 32 CPUs you can deploy 1 VM with 32 CPUs or 2 VMs with 16 CPUs each.

4.1. Scheduling BSP tasks

We implemented three different algorithms to schedule a set of BSP tasks. However, before scheduling BSP tasks, we need to pre-process tasks that uses more CPUs than available in each VM, reshaping then to use the number of available CPUs. This is done using Algorithm 1 from [Dutot et al. 2005]. From there, we schedule the tasks using one of the following algorithms.

Algorithm 1 Pseudo-code to reshape each task before scheduling

```
1: procedure RESHAPE_TASK($task$)
2:     if $task.number\_of\_cpus > cpus\_per\_vms$ then
3:         $ratio \leftarrow \lceil \frac{task.number\_of\_cpus}{cpus\_per\_vms} \rceil$
4:         $task.run\_time \leftarrow ratio * task.run\_time$
5:         $task.number\_of\_cpus = cpus\_per\_vms$
6:     end if
7: end procedure
```

The **first algorithm** is a simple *First In First Out* (FIFO) algorithm, allocating the tasks in order of arrival. While this algorithm is not really interesting as such, it serves as a base to analyze the algorithms in Section 4.2.

The **second algorithm** is the well-known *Largest Task First* (LTF), where *largest* refers to the task workload (*number of processors X task run time*), or, using a graphical representation, this would be the area of each task. In LTF we run Algorithm 1 from 1 up to the number of CPUs per VM to find the lowest generated makespan in each case, and select the lowest calculated makespan.

The **third algorithm** is called *Reduce Idle Time* (RIT), and its described in [Dutot et al. 2005]. This algorithm is composed of two steps:

1. Look for the *best task* such that, when schedule, the idle time is reduced or remains the same. *Best* means the smallest amount of idle time. We may apply the Algorithm 1 in each task before scheduling it.
2. If Step 1 fails, schedule the largest task not scheduled yet.

4.2. Distributing groups of tasks in each VM

Distributing groups of tasks in each VM only makes sense when there is more than one VM available to schedule these tasks. When there is only one VM, we use the scheduling resulting from the algorithms from Session 4.1 without any additional processing, and all algorithms listed in this session results in the same scheduling. However, if there are more than one VM, we need to distribute these tasks, preferably evenly between the different VMs. In this Section, we will present two algorithms to do this work.

The first algorithm is a simple *Round Robin* (RR) schedule, that takes the first task and schedule in the first VM, the second task and schedule in the second VM, and so on until there is no more VMs, when it starts from the first VM again, and continues until all tasks are allocated to one VM.

The second algorithm is named *Minimal Current Makespan* (MCM), a greedy algorithm that allocates the task in the VM that currently has the lowest makespan. The algorithm pseudo-code can be seen in Algorithm 2.

The algorithm tries to find the lowest increase of makespan by calculating how much the new makespan would be if the task was allocated in each existing VM. For this, in Line 6 we have a function that calculates what the makespan would be if the task was allocated in the current VM.

Algorithm 2 The minimal current makespan algorithm

```
1: procedure MINIMAL_CURRENT_MAKESPAN($tasks, vms$)
2:     for all $task$ in $tasks$ do
3:         $minimal\_makespan \leftarrow \infty$
4:         $best\_vm \leftarrow \emptyset$
5:         for all $vm$ in $vms$ do
6:             $aux \leftarrow calculate\_new\_makespan(task, vm)$
7:             if $aux < minimal\_makespan$ then
8:                 $best\_vm \leftarrow vm$
9:             end if
10:        end for
11:        $allocate\_task(task, best\_vm)$
12:    end for
13: end procedure
```

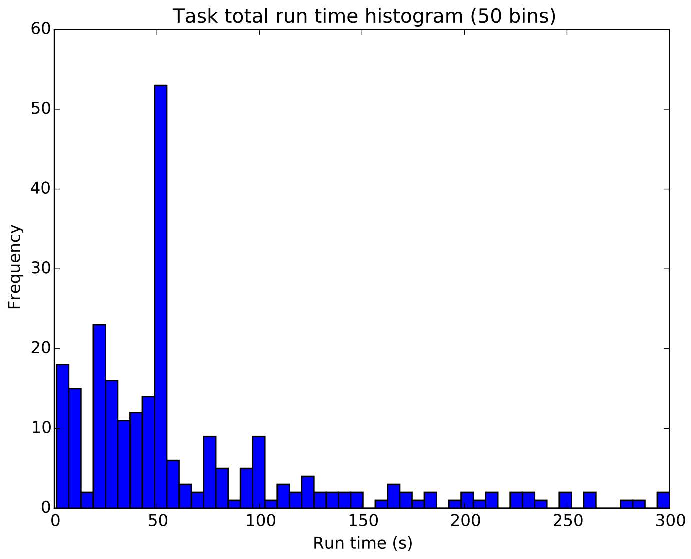

Figure 4. Histogram of tasks per total run time

5. Methodology

To simulate real workloads, we use the traces coming from [Feitelson 2015]. They are in a format that is easy to parse, as described in [Feitelson et al. 2014].

We selected the traces coming from The University of Luxemburg Gaia Cluster log from 2014[1]. This cluster is mainly used by biologists working with large data problems and engineering people working with physical simulations, typical tasks that would fit the BSP model. After parsing the tasks, we removed tasks with run time superior than 5 minutes (300 seconds), and selected the first 250 tasks since the number of tasks in trace is huge (composed of more than 50000 tasks). In Figure 4 is shown the number of times (*Frequency*) a task with total run time (or workload) appears in the filtered trace.

[1]`http://www.cs.huji.ac.il/labs/parallel/workload/l_unilu_gaia/index.html`

After that, we schedule the tasks using each possible combination of the developed algorithm schedulers described in Section 4 and calculated the expected makespan by running the resulting schedule in a simulator.

Before simulating these tasks in the cloud, we executed a program that measured how many loop cycles (x) are necessary to spend 1 second of CPU time inside the cloud. We simulate the execution of tasks by multiplying the resulting task run time after scheduling by x and running this tasks in multiple threads, equals to the number of processors used by the task.

So, if the task originally used 100 seconds in 4 CPUs, we are simulating these 100 seconds of CPU time by running 100 seconds of work inside the VM, in 4 different threads. Except that, thanks to different external factors, this value may decrease or increase during the experiment.

We executed these experiments in both Microsoft Azure and Google Compute Engine clouds. For these tests, we used VMs optimized for computing (Microsoft Azure `Standard G{2-5}` instances, Google's Compute Engine `n1-highcpu-{4-32}` instances). Each experiment was repeated 10 times, and the calculated standard deviation for population from each experiment was less than $\pm 1\%$.

The code used in these experiments is available in GitHub[2], under MIT License.

6. Results

In Figures 5, 7 and 9, we have a comparison between the measured makespan after running the tasks on Microsoft Azure (*Measured*) and the expected makespan from simulation (*Expected*), while in Figures 6, 8 and 10 we have the same for Google Compute Engine.

Just an observation about the graphics, when we use a VM with 32 CPUs, we only have one VM in these cases. So both *Minimal Current Makespan* and *Round Robin* algorithms returns the same scheduling, and the results are the same, as explained in the beginning of Section 4.2. But the results were plotted anyway to avoid confusion.

Starting with Microsoft Azure, the expected run time and the measured run time were closer than the result obtained in Google Compute Engine. Sometimes, the measured run time was faster than the expected run time. This can probably be explained because when there are idle CPUs, a task can run faster than expected. This should explain why only FIFO and RIT algorithms had this behavior, while in MCM, an algorithm that has better CPU utilization, did not have the same behavior.

In Google Compute Engine the measured run time is always slower than the expected run time, sometimes by a fair amount. This may be explained by Google's using more aggressive power consumption modes, or maybe Google is giving us logical threads (as in *Intel Hyper-Threading* technology) instead of physical cores. This may explain why we had worse performance when using a VM with more CPUs.

Comparing the task algorithm schedulers performance, LTF was the algorithm with the lowest run time, followed by RIT and FIFO, as expected by our simulations. When comparing the performance of algorithms to distribute the tasks between VMs,

[2] `https://github.com/m45t3r/cloudsched`

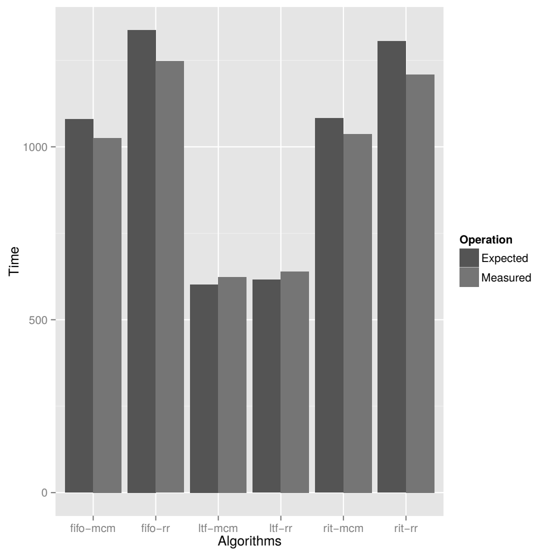

Figure 5. Makespan comparison in Microsoft Azure with 4 VMs X 8 CPUs

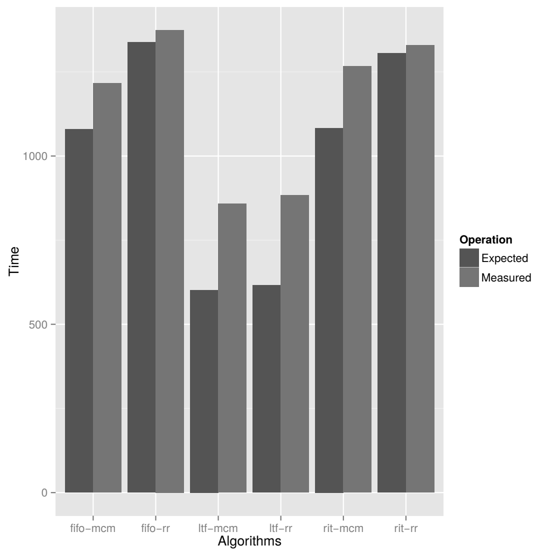

Figure 6. Makespan comparison in Google Cloud Engine with 4 VMs X 8 CPUs

however, the run time MCM was not significantly lower than RR, even getting a higher run time sometimes (combined with FIFO in Figure 7, for example). It is interesting to note that, sometimes, the expected run time of MCM is significantly lower than RR (compare RIT-MCM and RIT-RR in Figure 6, for example), but the measured run time is not. It may be thanks to the fact that MCM distribute the tasks more evenly between VMs, resulting in higher CPU utilization and, consequently, lower performance per task.

The overall standard deviation was very small (less than $\pm 1\%$). The tests were run over the course of a weekend, in different times, however even then we did not get too much variation in performance. This may be explained because of the use of VMs optimized for computation instead of standard VMs, or during our tests the utilization of other VMs in the cloud was low.

7. Conclusions and Future Work

Measuring performance in clouds is hard. Even when we did not have much performance variation between tests (we expected much higher standard deviations from our measurements), we still had distinct behaviors running the same tests in different cloud providers. Using VMs optimized for different use cases (general purpose, high CPU usage, etc.) seems to mean different things between public cloud providers, and even if we use a similar type of VMs between our tests, the results were still very different.

However, even when the expected run time is very different from the measured one, the performance was consistent. When the expected makespan for one scheduling is lower than another, the measured makespan is generally lower too. Our work shows that it is possible to study task scheduling algorithms in clouds, however it is necessary more tests and better heuristics to improve the accuracy of our scheduler, making the real run time match the expected run time more closely. This will be done in a future work.

Another future work, we will implement a *backfilling* strategy in both FIFO and

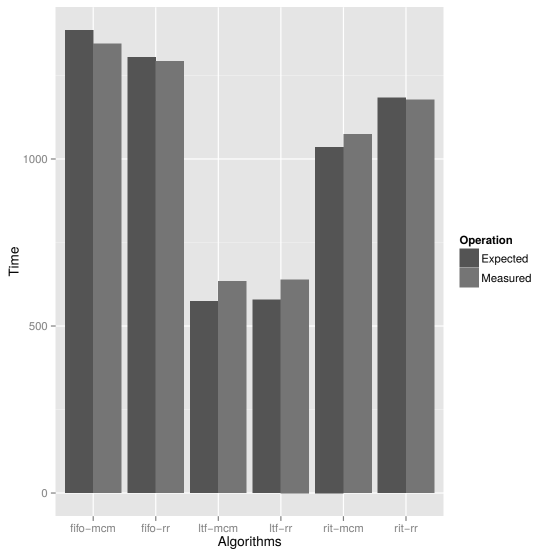

Figure 7. Makespan comparison in Microsoft Azure with 2 VMs X 16 CPUs

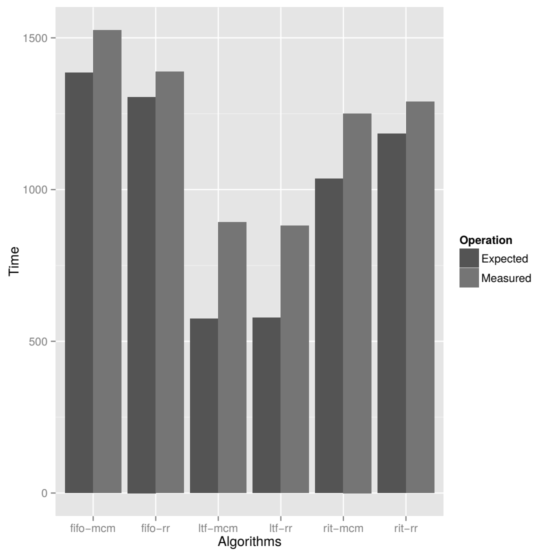

Figure 8. Makespan comparison in Google Cloud Engine with 2 VMs X 16 CPUs

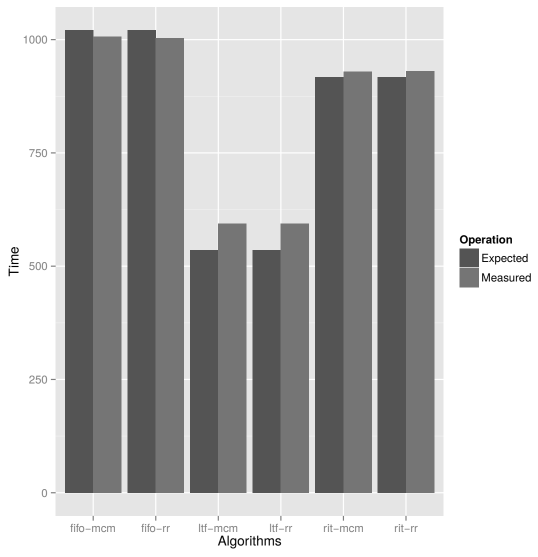

Figure 9. Makespan comparison in Microsoft Azure with 1 VM X 32 CPUs

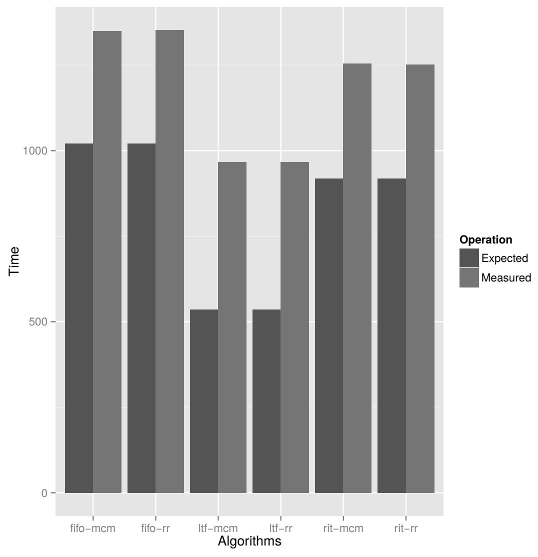

Figure 10. Makespan comparison in Google Cloud Engine with 1 VM X 32 CPUs

LTF algorithms [Mu'alem and Feitelson 2001]. Backfilling allows small tasks to fill gaps during the execution of tasks. This strategy helps to reduce the processor idle time and total makespan.

In addition, we will study how reshaping a BSP task might affect its performance. In this paper we used a linear model (reducing the number of processors increase the task run time linearly), but this is not necessary true for real parallel tasks, since in real parallel tasks have sequential parts and communication between processors.

Finally, we will expand this work to malleable BSP tasks. Since each BSP task has a defined state between each superstep, it is possible to stop execution of a task when it reach a superstep, checkpoint its state, and continue the execution using another set of resources.

Acknowledge

We would like to thank Microsoft and Google for the free access to their public clouds, also to CAPES and São Paulo Research Foundation (FAPESP) (procs. #2012/23300-7) for the fellowships.

References

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

Cáceres, E., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., Santoro, N., and Song, S. W. (1997). Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *Automata, Languages and Programming*, pages 390–400. Springer.

Calinescu, R. (1997). A BSP approach to the scheduling of tightly-nested loops. In *Proceedings of the 11th International Symposium on Parallel Processing*, IPPS '97, pages 549–553, Washington, DC, USA. IEEE Computer Society.

Chandio, A., Bilal, K., Tziritas, N., Yu, Z., Jiang, Q., Khan, S., and Xu, C.-Z. (2014). A comparative study on resource allocation and energy efficient job scheduling strategies in large-scale parallel computing systems. *Cluster Computing*, 17(4):1349–1367.

Cordeiro, D., Goldman, A., Kraemer, A., and Junior, F. P. (2013). Using the BSP model on clouds. *Cloud Computing and Big Data*, 23:123.

Dehne, F., Fabri, A., and Rau-Chaplin, A. (1993). Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the ninth annual symposium on Computational geometry*, pages 298–307. ACM.

Dehne, F., Hutchinson, D., Maheshwari, A., and Dittrich, W. (1999). Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 14–20.

Dutot, P.-F., Netto, M. A. S., Goldman, A., and Kon, F. (2005). Scheduling moldable BSP tasks. In *Job Scheduling Strategies for Parallel Processing*, pages 157–172. Springer.

Feitelson, D. (2015). Parallel Workloads Archieve. http://www.cs.huji.ac.il/labs/parallel/workload/. Accessed: 2015-07-30.

Feitelson, D. G., Tsafrir, D., and Krakov, D. (2014). Experience with using the parallel workloads archive. *Journal of Parallel and Distributed Computing*, 74(10):2967–2982.

Gibbons, P., Matias, Y., and Ramachandran, V. (1998). The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, 196(1–2):3–29.

Gonçalves, M., Cunha, M., Sampaio, A., and Mendonça, N. C. (2015). Inferência de desempenho: Uma nova abordagem para o planejamento de capacidade de aplicações na nuvem. *Anais do XXXIII Simpósio Brasileiro de Rede de Computadores e Sistemas Distribuídos*.

Huang, K.-C., Huang, T.-C., Tsai, M.-J., and Chang, H.-Y. (2014). Moldable job scheduling for HPC as a service. In Park, J. J. J. H., Stojmenovic, I., Choi, M., and Xhafa, F., editors, *Future Information Technology*, volume 276 of *Lecture Notes in Electrical Engineering*, pages 43–48. Springer Berlin Heidelberg.

Huang, K.-C., Huanga, T.-C., Chang, M.-J. T. H.-Y., and Tung, Y.-H. (2013). Moldable job scheduling for HPC as a service with application speedup model and execution time information. *Journal of Convergence*, 4(4):14–22.

Hunold, S. (2015). One step toward bridging the gap between theory and practice in moldable task scheduling with precedence constraints. *Concurrency and Computation: Practice and Experience*, 27(4):1010–1026.

Juurlink, B. H. H. and Wijshoff, H. A. G. (1998). A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271–318.

Kirtzic, J. S. (2012). *A Parallel Algorithm Design Model for the Gpu Architecture*. PhD thesis, Richardson, TX, USA. AAI3547670.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

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

Nuaimi, K. A., Mohamed, N., Nuaimi, M. A., and Al-Jaroodi, J. (2012). A survey of load balancing in cloud computing: Challenges and algorithms. In *Proceedings of the 2012 Second Symposium on Network Cloud Computing and Applications*, NCCA '12, pages 137–142, Washington, DC, USA. IEEE Computer Society.

Skillicorn, D. B. and Talia, D. (1998). Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169.

Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.

Wei, Y. and Blake, M. B. (2010). Service-oriented computing and cloud computing: Challenges and opportunities. *IEEE Internet Computing*, 14(6):72–75.

Yuvaraj, B. and Palanivel, K. A survey of virtual machine placement algorithms in cloud computing environment. *International Journal on Recent and Innovation Trends in Computing and Communication*, 3:121 – 126.