Scheduling BoT Applications in Grids using a Slave Oriented Adaptive Algorithm*

Tiago Ferreto[1], César De Rose[1], and Caio Northfleet[2]

[1] Faculty of Informatics - PUCRS, Brazil
ferreto@inf.pucrs.br, derose@inf.pucrs.br
[2] HP-Brazil
caio.northfleet@hp.com

Abstract. Efficient scheduling of Bag-of-Tasks (BoT) applications in a computational grid environment reveals several challenges due to its high heterogeneity, dynamic behavior, and space shared utilization. Currently, most of the scheduling algorithms proposed in the literature use a master-oriented algorithm, in which the master is the only responsible for choosing the best task size to send to each slave. We present in this paper a different approach whose main originality is to be slave-oriented, *i.e.* each slave locally determines, from a set of initial runs, which workload size is more adapted to its capacities and notifies the master of it. Finally, we show some measurements comparing our algorithm with other three well-known scheduling algorithms using the SimGrid toolkit.

1 Introduction

Computational grids as a platform to execute parallel applications is a promising research area. The possibility to allocate unprecedent amounts of resources to a parallel application and to make it with lower cost than traditional alternatives (based in parallel supercomputers) is one of the main attractives in grid computing. On the other hand, the grid characteristics, such as high heterogeneity, complexity and wide distribution (traversing multiple administrative domains), create many new technical challenges. In particular, the area of scheduling faces entirely new challenges in grid computing. Traditional schedulers (such as the operating system scheduler) control all resources of interest. In a grid, such a central control is not possible. First, the grid is just too big for a single entity to control. In a grid, a scheduler must strive for its traditional goals, improving system and application performance [1].

Bag-of-Tasks (BoT) applications are parallel master/slave applications whose tasks are independent to each other. A vast amount of work has been done in order to schedule efficiently Bag-of-Tasks applications improving the load balancing in distributed heterogeneous systems. Most of the algorithms focus on the adaptation of the workload during the execution, using either a fixed increment or decrement (*e.g.* based on an arithmetical or geometrical ratio) or a

* This research was done in cooperation with HP-Brazil.

more sophisticated function to adapt the workload. Yet the solutions presented are all based on some evaluation by the master of the slaves' capacities and of the tasks workload. This implies a significant overhead since the master has to maintain some kind of information about its slaves.

We propose in this paper the scheduling of BoT applications in Grids with a different approach whose main originality is to be slave-oriented, *i.e.* each slave locally determines, from a set of initial runs, which workload is more adapted to its capacities and informs the master of it. In turn, the master can compare the workload demanded by the slave to the network penalty paid and make the proper adjustments to adapt the workload. We have thus a workload adaptive algorithm.

2 Related Work

In this section we focus in self-scheduling algorithms [2]. These algorithms divide the total workload based on a specific distribution, providing a natural load balancing to the application during its execution. This class of algorithms is well suited for dynamic and heterogeneous environments, such as grids, and for divisible workload applications.

The Pure Self-scheduling [2] or Work Queue scheduling algorithm divides equally the workload in several chunks. A processor obtains a new chunk whenever it becomes idle. Due to the scheduling overhead and communication latency incurred in each scheduling operation, the overall finishing time may be greater than optimal [3].

The Guided Self-scheduling algorithm [4] (GSS), proposed by Polychronopoulos and Kuck, and Factoring [3], proposed by Flynn and Hummel, are based on a decreasing-size chunking scheme. GSS schedules large chunks initially, implying reduced communication/scheduling overheads in the beginning, but at the last steps too many small chunks are assigned generating more overhead [2]. Factoring was specifically designed to handle iterations with execution-time variance. Iterations are scheduled in batches of equal-sized chunks. The total size of the chunk per batch is a fixed ratio of the remaining workload.

In all algorithms shown above, the amount of workload sent to each slave is defined by the master. We propose in the following section another approach, where the evaluation of the load to be assigned to each slave is done by the slave itself.

3 Local Decision Scheduling Algorithm

The local decision scheduling algorithm (LDS) addresses BoT applications using divisible workloads, *i.e.* all independent tasks demands the same amount of computational resources. The algorithm focus on a heterogeneous, dynamic and shared environment, characterizing a typical computational grid. It is based on a distributed decision mechanism, building in each slave a performance model, which represents the application behavior based on resources utilization. Each

slave computes the task received, includes this information in its performance model and, based on the analysis of its performance model, calculates the best workload size to be computed at the next iteration.

The scheduling algorithm is divided in the following phases: setup, adaptive and finalization phases. The setup phase goal is to initialize and refine the performance model of each slave. The master sends tasks to slaves using a fixed quadratic increment. This process continues until it receives a signal from the slave in order to start the adaptive phase. This signal is generated when the performance model starts presenting estimates with minimum error.

The adaptive phase goal is to adapt this performance model if any variation is observed and to generate appropriate estimates of workloads size to be computed in the next iterations. The master side of the algorithm for the adaptive phase is presented in Algorithm 1. It sends to the slave a task using the fixed quadratic increment again, but at this time, it includes information about the time slice the slave has to compute at the next iteration ($execTime$) variable). This information is highly dependable on the application characteristics, workload (number of tasks), and environment conditions, and is currently static and manually defined. The master receives, after the processing of the task by the slave, the result, execution time of the task computed and an estimation of the next workload size in order to accomplish to the time slice defined at the master. At the next workload assignment for the slave, the master just changes the workload size to send to the slave accordingly to the estimate previously received. It keeps using this procedure until it reaches a specified limit (line 4). After this, it starts the finalization phase.

The slave side of the algorithm for the adaptive phase is presented in Algorithm 2. The slave starts a loop receiving tasks to be computed. Together with the task, it receives the workload size and execution time values. The workload is computed and its size with execution time inserted in a prediction table, which is used to compute the performance model. Using this prediction table and the execution time value received from the master, it computes the next workload size. After that, the slave sends to the master the result, execution time of the task received, and an estimation of the next workload size. The slave gets out from the loop when it receives a signal message from the master to initiate the finalization phase.

The finalization phase adjusts the workloads size computed in each slave in order to achieve load balancing, resulting in a better overall performance. When the master switches from the adaptive phase to the finalization phase, it stops using slaves' predictions and starts using the factoring algorithm till the end of tasks processing. After assigning the remaining tasks, the master starts a loop receiving the remaining results from the slaves.

3.1 Local Prediction of the Computational Load

In order to estimate the most suited workload, a slave needs a performance model for the execution of chunks of size $taskSize_i$. The model may include various data such as the execution time, memory utilization, cache access, etc, used to

Algorithm 1 LDS algorithm at master side

```
1: while there are tasks to schedule do
2:   for each available slave_i do
3:     execTime ⇐ maxExecTime
4:     if Σ_{i=1}^{numslaves} taskSize_i ≥ number of tasks remaining then
5:       start finalization phase
6:     else
7:       task ⇐ getTask(taskSize_i)
8:       send to slave_i the task, taskSize_i and execTime
9:     end if
10:  end for
11:  receive result, execTime and nextTaskSize
12:  taskSize_i ⇐ nextTaskSize
13: end while
```

Algorithm 2 LDS algorithm at slave side

```
1: while there are tasks to compute do
2:   receive task, taskSize and maxExecTime
3:   result ⇐ compute task
4:   insertPredictionTable(taskSize, execTime)
5:   nextTaskSize ⇐ predictNextSize(maxExecTime)
6:   send to master the result, execTime and nextTaskSize
7: end while
```

process a given task. In this preliminary version of our prototype we only take into account the execution time.

Given some N values $taskSize_1, taskSize_2, \ldots taskSize_n$ and the slaves data t (*e.g.* the execution time) the slave has to estimate $t(taskSize)$. In a multi-parameter model we could use algorithms such as the Singular Value Decomposition [5], one of the most robust for data modeling. It would fit the function t as a linear combination of standard base functions (*e.g.* $x \to e^x$, $\sqrt{}$, polynomials, $\ldots$).

Yet in the case where t only depends on the processor's speed, an affine model of the time required *vs.* the number of chunks to run is most realistic and used by other algorithms [6]. The modeling problem is therefore a basic linear interpolation problem of the measured running time $t_j, j = 1 \ldots n$ *vs.* the number of chunks $taskSize_j$. Beside the estimated coefficients a, b of the affine approximation $t = a + b \times taskSize$, the correlation coefficient is used to determine the correction of the interpolation and thus decide if more chunks should be sent in the initial phase, before entering in the adaptive phase.

The interpolation algorithm is very fast and thus does not prejudice the execution of the application. Moreover, it is trivial for a slave to determine the adapted task size, given the execution time t it has to run and the affine model (a, b). Note that in the case of a more complex, non-linear model, it would have to use a more time-consuming algorithm such as a gradient or dichotomic search to solve the $t = f(taskSize)$ equation.

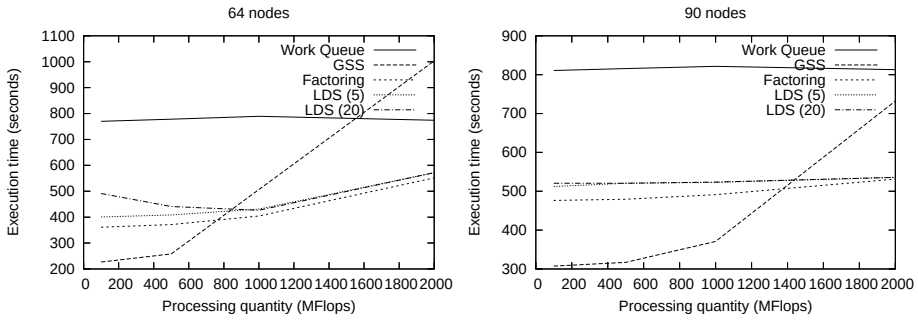

Fig. 1. Measurements scheduling 1000 tasks using 64 and 90 nodes.

4 Evaluation

We used the SimGrid [7] toolkit to evaluate our scheduling algorithm. The platform used for simulation is an example of grid model included in the SimGrid package. The model is composed by 90 heterogeneous machines connected by several links with different latency and bandwidth values.

We used this platform to simulate applications with different number of tasks (1000, 10000 and 100000 tasks) and quantity of computation per task (100, 500, 1000 and 2000 MFlop/s) using deployments with 64 and 90 nodes. In our experiments we assumed that communication costs to send one task to a slave is fixed (0.001 Mbyte/s) and to receive the result is irrelevant.

Each simulated application was executed using 4 different algorithms: Work Queue, Guided Self-scheduling, Factoring (using $\alpha = 2$) and LDS (using $\beta = 5$ and 20). LDS uses the β value to compute the $maxExecTime$ parameter, which is calculated dividing the estimate of the total execution time to compute all tasks sequentially by β multiplied by the number of slaves.

Figure 1 illustrates the measurements obtained for an application containing 1000 tasks begin executed in 64 and 90 nodes of the platform, with computation amount per task varying from 100 to 2000 MFlop/s. Using 64 nodes and computation quantity ranging from 100 to approximately 800 MFlop/s, the GSS algorithm presented the best results, after 800 MFlop/s the Factoring algorithm overcomed GSS. The same behavior is presented with 90 nodes, except that this transition is observed when computation quantity is approximately 1400 MFlop/s. The Work Queue algorithm presented the worst results in both simulations. The LDS algorithm presented a behavior similar to the Factoring algorithm in both simulations.

In Figure 2 the same measurements are presented for an application with 10000 tasks. The Work Queue algorithm presented again the worst results using 64 and 90 nodes. GSS, Factoring and LDS(5) presented similar values using low computation quantity (100 MFlop/s) and 64 nodes. Using tasks with more than 400 MFlop/s the LDS(5) and LDS(20) presented the best results. Using 90 nodes

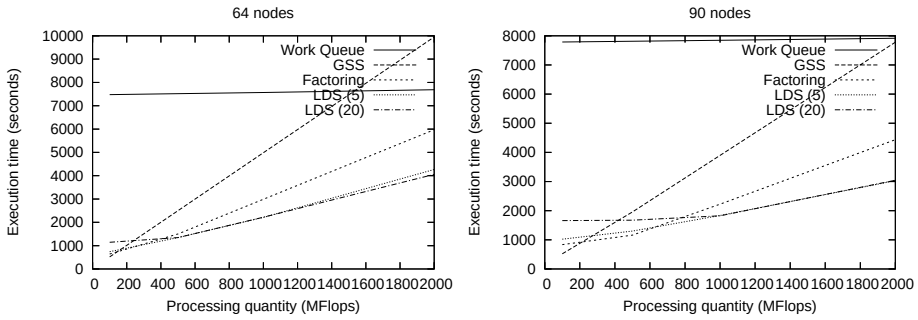

Fig. 2. Measurements scheduling 10000 tasks using 64 and 90 nodes.

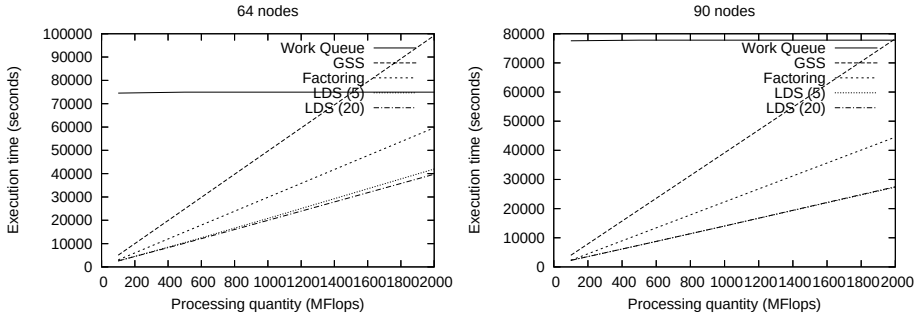

Fig. 3. Measurements scheduling 100000 tasks using 64 and 90 nodes.

and computation quantity ranging from 100 to approximately 200 MFlop/s the GSS algorithm presented the best results. Factoring behaved better from 200 to approximately 600 MFlop/s and after 600 MFlop/s, the LDS(5) and LDS(20) algorithm overcomed the other 3 algorithms.

The measurements for 64 and 90 nodes using an application with 100000 tasks (Figure 3) are very similar. The LDS algorithm presented the best results for all experiments and overcomed the Factoring algorithm in approximately 30%, *i.e.* the execution time of the application using the LDS algorithm was approximately 30% faster in comparison to the Factoring algorithm.

The measurements show that simple algorithms, such as, GSS and Factoring present good results when the total number of tasks and the computation quantity per task is low. With a higher number of tasks and computation quantity the LDS algorithm performs better, obtaining in some cases, a reduction of 30% in the execution time in comparison to the Factoring algorithm.

5 Conclusion and Future Work

In this paper we proposed a slave oriented adaptive algorithm for the scheduling of BoT applications in Grid environments. In this approach, each slave of a BoT application locally determines, based on a data modeling algorithm to evaluate its computational capacity on a received task, which workload is more adapted to its capacities, sending this information to the master. The main characteristic of our algorithm is that the computation of the best suited task size is entirely distributed, unlike other adaptive approaches.

We compared our algorithm with other well-known scheduling algorithms using the SimGrid toolkit and preliminary results indicate that this new approach behaves better in several of the test cases. The best results where obtained when the number of tasks is high (100000 tasks), the computation amount of each task is also high (1000 and 2000 MFlop/s), and the Grid is composed of several heterogeneous resources (90 nodes) resulting in a mean performance increase of approximately 30% over the Factoring algorithm.

The main limitation of our algorithm currently lies in the modeling of the slave's capacities to treat the master's tasks. We intend to improve the data modeling and the possibility to extrapolate the model for values of the workload that could be less regular. In this initial work the slave only evaluates its CPU performance. A direct extension of the algorithm would be to include a local evaluation of the slave's memory usage. We also face the inclusion of a historical log maintained by the slave about its availability so as to let it require a workload most adapted to the time-frame it knows it can work for the master.

Nevertheless we believe this novel approach is promising and already a very good alternative to be considered when a scheduling algorithm is needed for BoT applications in Grids.

References

1. da Silva, D.P., Cirne, W., Brasileiro, F.V.: Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In: Euro-Par 2003. Volume 2790 of Lecture Notes in Computer Science., Springer (2003) 169–180
2. Chronopoulos, A.T., Andoine, R., Benche, M., Grosu, D.: A CLass of Loop Self-Scheduling for Heterogeneous Clusters. In: Proceedings of CLUSTER'2001. (2001)
3. Hummel, S.F., Schonberg, E., Flynn, L.E.: Factoring: A Method for Scheduling Parallel Loops. Communications of the ACM **35** (1992) 90–101
4. Polyhronopoulos, C.D., Kuck, D.: Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. IEEE Trans. on Computers **36** (1987) 1425–1439
5. Press, W.e.a.: Numerical Recipes in C: The Art of Scientific Computing. Number ISBN 0521431085. Cambridge University Press (1993)
6. Beaumont, O., Legrand, A., Robert, Y.: Scheduling divisible workloads on heterogeneous platforms. Parallel Computing **29** (2003) 1121–1152
7. Casanova, H.: Simgrid: A toolkit for the simulation of application scheduling. In: Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'01). (2001)