Dynamic Grid Scheduling Using Job Runtime Requirements and Variable Resource Availability

Sam Verboven, Peter Hellinckx, Jan Broeckhove,
and Frans Arickx

CoMP, University of Antwerp, Middelheimlaan 1,
2020 Antwerp, Belgium
Sam.Verboven@ua.ac.be

Abstract. We describe a scheduling technique in which estimated job runtimes and estimated resource availability are used to efficiently distribute workloads across a homogeneous grid of resources with variable availability. The objective is to increase efficiency by minimizing job failure caused by resources becoming unavailable. Optimal scheduling will be accomplished by mapping jobs onto resources with sufficient availability. Both the scheduling technique and the implementation called PGS (*Prediction based Grid Scheduling*) are described in detail. Results are presented for a set of sleep jobs, and compared with a first come, first serve scheduling approach.

1 Introduction

During the last couple of years, the demand for more computational resources within grid systems has grown significantly. This has lead to situations where traditional dedicated grids can no longer satisfy that demand. In an effort to increase the amount of available resources one can turn to the concept of *CPU harvesting* on idle resources. Organizations and private users all over the world have large amounts of user-controlled resources (e.g. workstations, personal computers) that often go unused over long periods of time. These resources could be made available to heavily used grid systems without the need to invest in extra equipment and infrastructure. Some existing grid systems such as Condor [1], Globus [2] and CoBRA [3][4] allow user-controlled resources and dedicated clusters to co-exist within a single grid.

There are some drawbacks that need to be considered. Job failure caused by a resource becoming unavailable becomes an issue when employing large numbers of non-dedicated resources. In [5] a performance analysis of the impact of failures is made for a range of scheduling policies. It concludes that failures have a large impact on the total runtime of failure unaware scheduling policies. The author presents a solution using temporal information in combination with checkpointing [6]. However, this approach requires that users instrument their code to enable the checkpointing. For example, in Condor users have to link their code to a system call library to allow for checkpointing. Such instrumentation is not always possible and, in any case, it represents an additional burden that

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 223–232, 2008.

users have to assume in order to use the grid. Users should be unaware of the scheduling technique being used, and the complexity of distributing a problem needs to be minimized. Since we want the user to have as few specific requirements as possible, a more general solution is proposed. Such implementation however requires more information than a classic grid scheduling technique. The extra information required is time related: how long does it take for a certain jobs to execute on a particular resource and for how long will this resource be available.

Availability of resources can be divided into two categories, *predicted availability* and *planned availability*. Planned availability could be used in the case of workstations that are periodically available e.g. during non-work hours, or dedicated grid resources that are periodically unavailable e.g. during maintenance. Predicted availability presents a more complex problem. Predictions have to be made by monitoring the grid system. Methods have already been developed in specific areas, such as Fine-Grained Cycle Sharing systems [7], as well as more general approaches suitable for both desktop and enterprise environments [8]. In this paper we will concentrate on a scheduling algorithm based on known limited availability information, e.g. resources configured to become (un)available at predetermined times. These situations can be found in large organizations where workstations are available during nights and weekends or even lunch hours. We will use the term *uptime* to define the amount of time a resource is available for distributed computations.

The other crucial element needed to efficiently schedule, is the job runtime. In general one distinguishes three possible techniques to predict job runtimes: *code analysis* [9], *analytical benchmarking/code profiling* [10] and *statistical prediction* [11]. The best method for predicting runtimes depends heavily on the type of application one is distributing.

In this paper will propose a *fault-aware scheduling* mechanism for use with *Bag-of-Tasks* (BoT) applications [12], loosely-coupled parallel applications consisting of mutually independent jobs. Using BoT applications removes inter-job constraints when scheduling. The proposed technique is based on the availability of job runtime estimates and resource availability estimates. Instead of just tolerating failures like *fault-tolerant scheduling* this technique will pro-actively try to prevent failures from occurring. By distributing jobs only to resources available for the full executing time of the job, no CPU cycles are wasted on jobs that will be unable to complete. For the implementation and testing of PGS, the CoBRA [3][4] grid system is used.

The rest of the paper is organized as follows. In section 2 an overview is given of the scheduling technique. We discuss collecting the needed information and the scheduling mechanism. Section 3 presents the implementation of the scheduling technique introduced in section 2. A short introduction is given into CoBRA followed by the implementation details. In section 4 a description is given on the testing technique followed by an overview of the test results. More information on future work can be found in section 5 followed by a conclusion in section 6.

2 Scheduling Technique

The goal is simple, given the available information try to find the optimal job distribution that causes as little job failure as possible while minimizing the total application runtime. This goal can be achieved by insuring no job is scheduled onto a resource that will not be available for the required runtime. To make the scheduling mechanism possible, the required information must first be gathered. This information consists of two main parts, the time needed by a given job to complete itself on a certain resource, and the amount of time this resource will remain available. The way the job runtimes are predicted may depend on the type of jobs. Similarly, the expected uptimes can also be obtained in various different ways depending on the grid configuration. As such, our scheduling technique needs to be independent of the way this information is determined. Care needs be taken when gathering initial data and consequent updates of this data. Each update constitutes data transfer across the network that could form a possible bottleneck slowing down the scheduling process.

2.1 Collecting Job Runtime Information

The first requirement is efficiently obtaining accurate job runtime predictions. Since large amounts of jobs may be available to the scheduler at any given time, the number of network messages required per job needs to be reduced to a minimum. A simple yet elegant solution is proposed: jobs are submitted to the scheduler once a stable runtime prediction is available. By excluding functionality to gather runtime updates, dependent on the prediction method, the number of needed network messages is reduced. There are, however, drawbacks if a prediction method is used that relies on information from previous jobs. In this case, accurate job runtime estimates can not be made from the start and job submission becomes an iterative process. To solve this problem an optional interface is added which allows the prediction component to update job runtimes even after submission to the scheduler. When necessary, this allows updates to be made when new information becomes available.

2.2 Collecting Resource Availability Information

For the second requirement, obtaining accurate resource uptimes, more effort is needed. When a resource becomes available and its uptime is given we cannot assume that this value remains constant. Updates at regular intervals are needed. Between these updates the uptime prediction is estimated using a standard interpolation technique. This is not the only factor that needs to be taken into consideration. The scheduling mechanism will attempt to match job runtimes with the remaining resource uptimes, taking into account the queue of jobs already associated with that resource. These jobs can be found in two locations: on the resource after being submitted and in a local queue on the scheduler after being scheduled for the resource but not yet submitted. This requires information records containing two additional aspects next to the estimated uptime:

1. The predicted runtimes for the jobs that have been scheduled to a particular resource but have not yet been submitted.
2. The predicted runtime of the currently running job and the already elapsed runtime.

With this information it is possible to accurately predict tl, the time a resource has left for computations :

$$tl = ru - (et_{rj} - el_{rj}) - \sum_{i=0}^{qj} et_i$$

With ru the resource uptime, et_{rj} the runtime of the current running job, el_{rj} the elapsed runtime, qj the amount of jobs queued on the scheduler for this resource and et_i the runtime of the i'th job in this queue.

2.3 Scheduling Mechanism

When the required information has been obtained the scheduling mechanism can start mapping jobs onto resources. It requires three lists to be maintained. The list of all available resources, a list of all jobs that need to be run but have not yet been scheduled, and a list of jobs that have already been processed in some way but are still in need of scheduling (e.g. a job failed or no suitable resource was found in a previous mapping attempt). This last list is called the priority queue, and in each scheduling step we first try to map the jobs in this queue to a resource before turning to the regular job queue. In an effort to maximize resource utilization the main scheduling algorithm is split up into two complementing parts.

The first part consists of finding and scheduling a first job for each resource. When there are large amounts of resources in the grid, the time needed by the second part might leave some resources without a job to execute. To prevent this, the scheduler makes sure each resource is given at least one job for its queue before filling all queues with the most appropriate jobs. By taking this first step, as little resources as possible are wasted while running the next scheduling step. The algorithm used in the first step works as follows:

- Select resource R with an empty queue from the available resources.
- Select the first job J so that runtime J $<$ uptime R.

These steps are repeated as long as a resource R is found. In this first part of the scheduling mechanism the mapping is resource oriented so as to increase the throughput.

The second part of the mapping is done by iterating over the jobs and matching them to resources. For this part, two requirements need to be fulfilled. There have to be resources with sufficient uptime left and unfilled queues, and there have to be jobs left to execute. As long as this requirement is met we keep matching jobs to resources and adding them to their queues. After a first job has been put in each resource's queue, all resources are ordered by the amount of

computing time they have left. Using this ordered list, jobs are then be placed on the resource with the lowest amount of remaining uptime strictly greater than the required job runtime. This way resources with time to execute longer jobs are available for longer jobs further down the queue.

The jobs that have runtimes that are too long are put on the priority queue. When the resource list is updated and new resources are added with longer available uptimes, these longer jobs will get the first chance to be submitted. This method allows longer jobs to be run as soon as possible while maximizing resource utilization by filling the gaps with smaller jobs. It is important to notice that this second scheduling step is job oriented and tries to minimize the round trip time.

3 Implementation

3.1 CoBRA

The CoBRA grid [3] stands for *Computational Basic Reprogrammable Adaptive grid* and can be defined as:

> A portable generalized plug-in middleware for Desktop computing that allows developers to dynamically adapt and extend the distributed computing components without compromising the functionality, consistency and robustness of the deployed framework.

We have opted for the CoBRA middleware, as it allows to easily extend and dynamically replace available middleware components. This makes it a perfect environment to implement and test a new scheduling mechanism. It also provides a standard scheduler whose functioning can be compared with the newly proposed technique. The standard CoBRA grid scheduler uses a first come, first serve (FCFS) method to distribute jobs across available resources. The only requirement for a job to be scheduled on a resource is that the resource queue is not full. A list of available resources is composed using a resource manager component. A resource proxy, through which jobs can be submitted, is returned. Each job is taken from the queue and submitted to the first resource that has less jobs than the maximum configured queue length. When a resource fails and jobs are rescheduled they are added to the front of the queue.

3.2 Extending the Scheduler

This section gives a short overview of the implementation details. It describes how the CoBRA grid components are extended to obtain the functionality described in section 2. As can be seen in figure 1, the scheduling implementation consists of four main components. The *JobContainer* objects whose main purpose is to provide a local cache for remote information. The *ResourceContainer* objects are also used for caching purposes but, more importantly, maintain the job queues during job scheduling and submission. A separate *JobSubmitter*

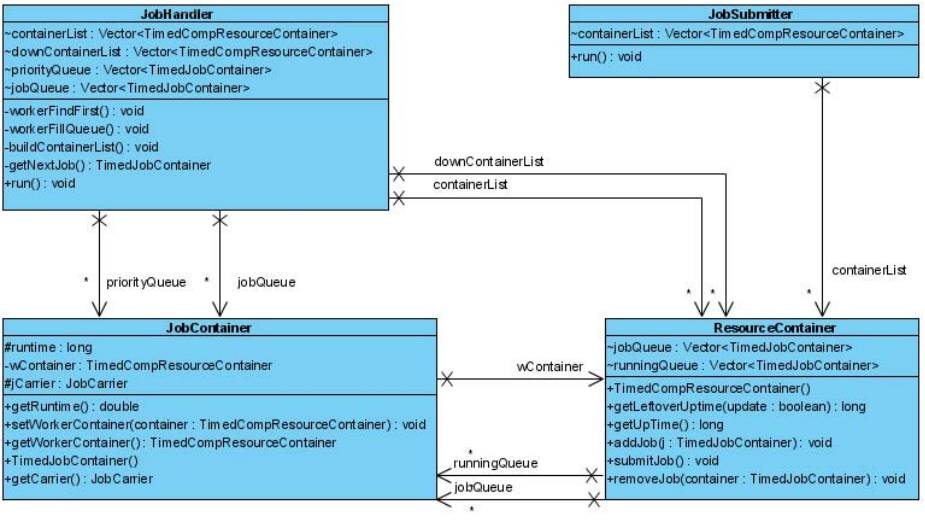

Fig. 1. Simplified scheduling UML diagram

thread is used to continuously submit jobs from the local queues on the scheduler to the resources. Combined with the two step scheduling mechanism this minimizes the delay between deciding where a resource should be submitted and the actual submission. The most important part of the scheduling technique is implemented in the *JobHandler* thread. Here all required information is gathered, organized and subsequently used to distribute each job to its most suitable resource. The JobHandler thread operates in three main steps:

1. Building/updating the list of JobContainer objects encapsulating the currently available resources.
2. Finding a first job for each resource that currently has an empty queue.
3. Filling the queues.

The JobContainer list built in step 1 is made up of two separate parts: the currently available resources and the previously available resources. When a resource becomes unavailable its container is moved to the previously available list and all its queued jobs are added to the *priorityQueue*. The subsequent steps are implemented as previously described in section 2.

4 Testing and Results

4.1 Testing Technique

To facilitate easy, correct and comprehensive testing, we use the "sleep testing technique" to evaluate the implemented scheduling mechanism. This technique was first introduced in [13] and has already been used for testing in CoBRA

[3]. It allows to generate many different simulated workloads using a limited number of parameters. It is also possible to simulate real, recorded workloads by replacing the actual jobs with sleep jobs that occupy the resource for the same amount of time. An additional benefit is that we know the exact job runtime and that it is independent of the machine type the job is executed on.

The next element needed to benchmark the proposed scheduling technique is a set of resources that have to become periodically unavailable. The requirement for these resources is that they become available and go down for predictable amounts of time. A practical solution that takes advantage of the CoBRA grid philosophy is used. All different components of the CoBRA grid consist of pluglets registered in a central lookup service, this includes the worker pluglets deployed on the resources. By encapsulating the existing worker functionality inside a new pluglet responsible for starting and stopping the worker, we can configure the availability of the corresponding worker resource. The encapsulating pluglet reads a startup-configuration file containing pairs of integers. These pairs contain the amount of seconds a resource will be available followed by the amount of seconds the resource will be unavailable. This way we can retain the original worker functionality while still having a reliable and accurate way of obtaining the availability information by simply requesting it from the pluglet. The reuse of the original worker allows for a more accurate comparison between old and new system tests.

4.2 Test Configuration

To test the proposed scheduling technique three different job scenarios are used. Each scenario is composed of a series of increasingly larger sleep jobs totaling 482 minutes. By changing the job duration and the amount of jobs we can test the impact different types of workload have on both the PGS and the original FCFS scheduling approach. Intuitively, in an environment where resources frequently become unavailable, small jobs should have less impact on the total runtime. The test scenarios describe the following job configurations:

1. 960 jobs ranging from 0.25 to 60 seconds in steps of 0.25 seconds (4 of each).
2. 480 jobs ranging from 0.5 to 120 seconds in 0.5 seconds steps (2 of each).
3. 240 jobs ranging from 1 to 240 seconds in 1 second steps.

The grid configuration used consists of three major components: 1 broker (Intel Pentium 4 CPU @ 2.26GHz, 512 MB), 8 resources (Intel Pentium 4 CPU @ 2.26GHz, 512 MB) and 1 scheduler (Intel Core2 CPU 6400 @ 2.13GHz, 1 GB). The resources are configured with the uptimes given in table 1. Each resource

Table 1. Uptime of the resource at restart

Resource	1	2	3	4	5	6	7	8
Uptime (Seconds)	84	117	163	228	318	443	619	864

is restarted after 1 second. On average it takes 1.2 seconds for a resource to become available again, including the time needed for the startup. Taking into account the total problem time of 482 minutes spread over 8 resources, the lower boundary for the total run time is 60 minutes and 15 seconds.

4.3 Results

Each test is run ten times and an average is taken from these runs. For comparison we add a scenario 0, corresponding to scenario 1, in which resources remain available during the full test run. The next three test scenarios use resources with the uptimes described in table 1.

In table 2 and figure 2 we show the test results comparing the standard FCFS approach with the PGS approach. It is observed that PGS is more efficient, with a more pronounced difference as job times (in terms of runtime) increase: for job configuration 1 the difference is 13.31% on average, increasing up to 38.64% for configuration 3. The reason for this lies in job failures that occur more frequently with FCFS. As average job times become longer, the amount of lost CPU cycles increases accordingly when failure occurs. This increases the relative benefit obtained by using PGS. From table 1 we can conclude that the standard deviation remains relatively small which proves the consistincy of the obtained results.

Table 2. Comparison between PGS and FCFS

Scenario	0	1	2	3
PGS (Minutes)	63.54	65.54	68.46	87.88
FCFS (Minutes)	63.21	74.26	87.3	121.83
PGS STDEV	0.19	0.39	073	2.03
FCFC STDEV	0.12	0.63	1.68	2.25
Difference	-0.51%	13.31%	27.51%	38.64%

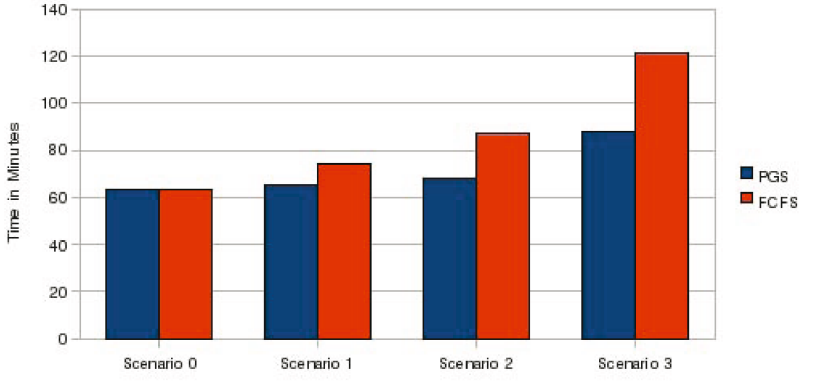

Fig. 2. Test results using the job configurations

5 Future Work

Future work can be performed in two directions, improving the scheduling technique and introducing more realism in the tested grid implementation. For this paper we have chosen to work with sleep jobs, allowing the sole focus of the tests and results to be on the scheduling mechanism. It is however not realistic to assume job runtimes prediction to be 100% accurate. Work is already underway to further extend the CoBRA grid with a system to automatically generate predicted runtimes for parameter sweep applications. The idea is to add a pluglet to the grid which is a front-end for the GIPSy *ModelBuilder* [14] that has already been developed by our research team. This front-end can then be used to generate predictions for jobs contained in a particular application. While runtime data is gathered the ModelBuilder will continuously keep building better models and suggest sets of points for which data is preferably obtained. The newest model is used to update job runtime predictions and reorder the queue of jobs that still need scheduling. This ensures the most valuable runtime data is gathered as soon as possible.

6 Conclusion

In this paper, we proposed a dynamic fault-aware scheduling mechanism that uses job runtime predictions and resource availability predictions to improve performance of BoT applications. This technique was implemented and compared to the FCFS scheduling technique. Empirical results show that large reductions in total runtime can be achieved in situations with variable resource availability. The difference in total runtime increases in favor of the proposed mechanism as job runtime increases. Results also indicate that the scheduling overhead remains the same in situations where resources are continuously available.

References

1. Condor, `http://www.cs.wisc.edu/condor/`
2. Globus, `http://www.globus.org/`
3. Hellinckx, P., Arickx, F., Broeckhove, J., Stuer, G.: The CoBRA grid: a highly configurable lightweight grid. International Journal of Web and Grid Services 3(20), 267–286 (2007)
4. Hellinckx, P., Stuer, G., Hendrickx, W., Arickx, F., Broeckhove, J.: Grid-user driven grid research, the CoBRA grid. In: CCGRID 2006: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2006), Washington, DC, USA, p. 49. IEEE Computer Society, Los Alamitos (2006)
5. Zhang, Y., Squillante, M.S., Sivasubramaniam, A., Sahoo, R.K.: Performance implications of failures in large-scale cluster scheduling. LNCS. Springer, Heidelberg (2005)
6. Litzkow, M., Tannenbaum, T., Basney, J., Livny, M.: Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department (April 1997)

7. Ren, X., Lee, S., Eigenmann, R., Bagchi, S.: Resource availability prediction in fine-grained cycle sharing systems. In: Proceedings of the Conference on High Performance Distributed Computing (2006)
8. Brevik, J., Nurmi, D., Wolski, R.: Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems. In: CCGRID 2004: Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid, Washington, DC, USA, pp. 190–199. IEEE Computer Society, Los Alamitos (2004)
9. Reistad, B., Gifford, D.K.: Static dependent costs for estimating execution time. SIGPLAN Lisp Pointers VII(3), 65–78 (1994)
10. Yang, J., Ahmad, I., Ghafoor, A.: Estimation of execution times on heterogeneous supercomputer architectures. In: ICPP 1993: Proceedings of the 1993 International Conference on Parallel Processing, Washington, DC, USA, pp. 219–226. IEEE Computer Society, Los Alamitos (1993)
11. Iverson, M.A., Özgüner, F., Potter, L.C.: Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. In: HCW 1999: Proceedings of the Eighth Heterogeneous Computing Workshop, Washington, DC, USA, p. 99. IEEE Computer Society, Los Alamitos (1999)
12. Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauve, J., Silva, F.A.B., Barros, C.O., Silveira, C.: Running bag-of-tasks applications on computational grids: The MyGrid approach. ICPP 00, 407 (2003)
13. Hellinckx, P., Stuer, G., Dewolfs, D., Arickx, F., Broeckhove, J., Dhaene, T.: Dynamic problem-independent metacomputing characterization applied to the condor system. In: Procedings ESM 2003, pp. 262–269 (2003)
14. Hellinckx, P., Verboven, S., Arickx, F., Broeckhove, J.: Scheduling parameter sweeps in desktop grids using runtime prediction. Poster, Grid@Mons (2008)