ReGS: User-level Reliability in a Grid Environment

José Afonso Lajas Sanches, Patrícia Kayser Vargas,
Inês de Castro Dutra, Vítor Santos Costa
COPPE/Sistemas, UFRJ, Rio de Janeiro, RJ, Brazil
{sanches, kayser, ines, vitor}@cos.ufrj.br

Cláudio Fernando Resin Geyer
Instituto de Informática
UFRGS, Porto Alegre, RS, Brazil
geyer@inf.ufrgs.br

Abstract

Grid environments are ideal for executing applications that require a huge amount of computational work, both due to the big number of tasks to execute and to the large amount of data to be analysed. Unfortunately, current tools may require that users deal themselves with corrupted outputs or early termination of tasks. This becomes incovenient as the number of parallel runs grows to easily exceed the thousands. ReGS is a user-level software designed to provide automatic detection and restart of corrupted or early terminated tasks. ReGS uses a web interface to allow the setup and control of grid execution, and provides automatic input data setup. ReGS allows the automatic detection of job dependencies, through the GRID-ADL task management language. Our results show that besides automatically and effectively managing a huge number of tasks in grid environments, ReGS is also a good monitoring tool to spot grid nodes pitfalls.

1. Introduction

Experiments in areas such as physics, machine learning or bioinformatics require a huge amount of computational work, both due to the big number of tasks to execute and to the large amount of data to be analysed. One good example are applications in machine learning, which often need to repeat experiments on different data subsets [2]. Often, as the size of the problem increases, the size of the data set will grow very quickly, requiring a huge number of experiments and computational jobs – sometimes with high execution times.

Performing such jobs in an uniprocessor environment would require months or even years, and in practice would make their realization impossible. Thus, *parallelism* must be used. Namely, *Grid environments* have become a popular alternative, because they make a huge amount of computing power available [5].

Grid environments bring new challenges. Most often, the resources in a grid are outside the bounds of a local network, so that their utilization might depend on accounting, authorisations, and communication speed. If the application requires a large number of jobs, it is often the case that some of those jobs will be lost. Even worse, because of the heterogeneity of grid environments, it is not impossible that jobs may return erroneous output (eg, due to corruption in file operations, network flooding, or to bugs in one of the software environments involved in the execution). Given a large number of jobs, it is extremely difficult to deal with those incidents case by case, which quite often requires resubmitting the corrupted jobs.

Unfortunately, local resource schedulers in grid environments, like Condor [10] and SGE [3], do not support automatic resubmission of corrupted tasks, leaving this task to the user. Available application managers, such as Chimera [6] and the Condor DAGMan, focus on solving the problem of automatic detection of *dependencies* between jobs. Towards building an integrated software to control and monitoring applications with large numbers of jobs, Dutra [2] *et al.* built a prototype which could prepare inputs for jobs automatically, and which further included a daemon to detect corrupted jobs. The prototype showed a need for user control: users should be able to easily check whether some runs are having problems. It was also found that in some applications a few tasks might be "problematic" and actually require user intervention. Ideally, we would like to do as much as possible automatically, and require actual user intervention only when it is clearly needed.

ReGS (Reliable Grid Submission) addresses theses issues by providing an interactive framework for job monitoring. First, ReGS relies on a user-friendly web interface, by which the user can enter parameters for his/her experiments, and interactively assess application execution and results. Second, ReGS includes an advanced job dependency framework, which allows for application control being transparent. ReGS automatically receives and prepares the input data, converts the tasks to a task management lan-

guage, GRID-ADL, calls the GRID-ADL parser, and makes the interface between the graph generated by the parser and the local resource manager residing on a grid node.

Following the Grid philosophy, we try to take best advantage of currently existing tools. Therefore, in our experiments we rely on a grid environment that uses Condor as resource manager, and convert our application graph to the DAGMan language. We use an application in the context of machine learning, which is detailed further in Section 3.

Our text is structured as follows. Section 2 gives the fundamentals we rely for our work. In Section 3, the purpose and implementation of our software are presented and detailed. In Section 4, we describe our experiments, and discuss results. Lastly, we present our main conclusions and discuss future work.

2. Background

In brief, a grid provides *heterogeneous* computational resources in a *large-scale* way. Grid applications range from distribute computing to collaborative computing [4]. Our interest focus on high-throughput computing, a particularly important set of applications where a grid may be called to process thousands of related jobs (or processes). Total execution time may span several days. In this section, we briefly discuss state-of-the art tools available for high-throughput computing in grid environments.

Job Schedulers match users' *jobs* to the available *resources*. It is unsurprising that many of the grid's job schedulers were initially developed for local distributed platforms. Some of the best known job schedulers are: Condor [10], PBS [8], SGE [3] and LSF [1]. We shall focus on Condor, as Condor is openly available, well supported, widely used in grid environments, and benefits from *opportunistic scheduling* - that is, it can exploit idle CPU cycles.

The *Condor High Throughput Computing System* [10] is a system for large-scale job and resource scheduling. Condor is designed to perform well in two areas: *high-throughput computing* and *opportunistic computing*. These areas are tightly related; one can obtain great utilizable computing power, if there are ways to make use of idle CPU cycles in computers. Some of the special mechanisms included in Condor are the following:

- *ClassAds* - ClassAds are similar to newspaper ads; each machine in a Condor pool informs about its attributes (such as RAM memory, CPU type, operating system, etc), through a ClassAd. Similarly, when a job is submitted, a user specifies the requirements needed for the job's execution by means of a ClassAd.
- *Checkpointing and migration* - If a machine in a pool is suddenly unavailable, the job may be *checkpointed* - that is, to record the point where the job stopped - and migrate to other available machine.
- *Remote System Calls* - Although Condor migrate jobs to remote machines, it can keep the local execution environment, thanks to remote system calls.

Automatic Job Management One important issue is respecting *dependencies* between jobs. If the number of jobs is large, it becomes difficult to control manually what jobs to start when a job finishes. Recent work in this direction includes:

- Chimera [6] is a system to manage data which is *derived* from other data (so-called *virtual data*), using the VDL language. Among other functions, this language can track dependencies between jobs by analyzing their input and output files. Chimera is able to generate a directed acyclic graph (DAG) from the dependencies, with the help of the Pegasus planner [7].
- Condor DAGMan is a meta-scheduler for Condor. As Condor does not schedule jobs based on dependencies, the Condor DAGMan can do it in an order represented by a DAG.

Because the DAGMan's language is directly understood by a Condor environment, we decided to use it in our work and discuss it next.

The *Condor DAGMan* allows the user to control possible dependencies among the jobs by means of a special script language. The tool does allow new attempts of resubmission through a special `RETRY` command (only if the error was something like a unmounted filesystem – events like network errors are not covered by DAGMan). However, if the number of the user's jobs is high, say hundreds, it will be impracticable to identify each job and to settle each dependency by writing a script in DAGMan's language. If a job produces an error as a missing or corrupted output file, DAGMan will not recognize it unless the user explicitly provides a `POST` program in the script, which will activate a new job written by the user to check the job's outputs. Finally, there is no time limit for a job to stay in Condor's job queue; a time limit could be useful if a deadlock occurs.

3. The ReGS Tool

The name ReGS stands for *Re*liable *G*rid *S*ubmission. ReGS is designed to reliably run experiments that spread a large amount of jobs, and is represented in Figure 1. ReGS consists of three modules, which communicate through a data file (submission) and through a status database:

- *Web Interface* - It provides the user with easier access to the grid environment and with information about the jobs' executions, via HTML pages. In Figure 1 this

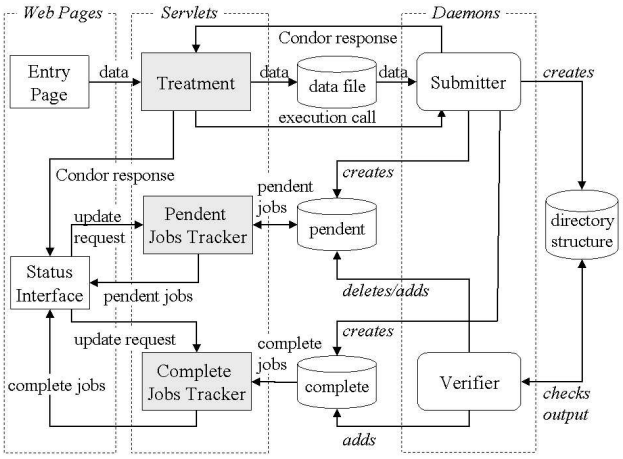

Figure 1. ReGS main components

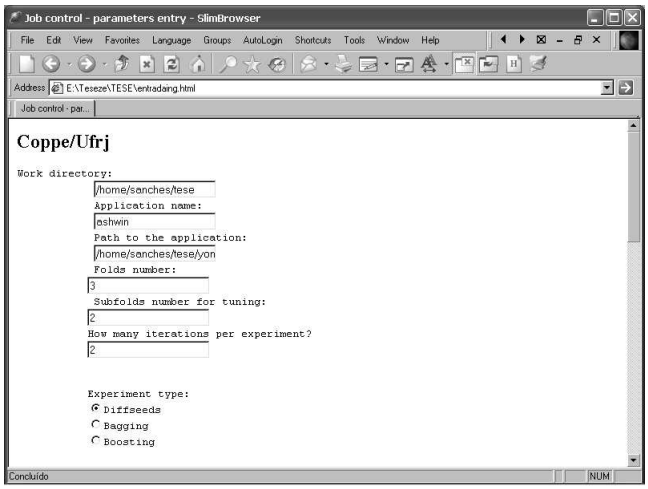

Figure 2. The *Entry Page*

corresponds to the set of Java servlets, represented as gray boxes. Each servlet is concerned with a specific task.

- *Job Submission* - the *submitter* prepares input data for jobs, and submits the jobs to the grid environment.
- *Job Monitoring* - the *verifier* collects and organizes information about the jobs, and verifies how their execution is proceeding - the tool will resubmit the jobs if it becomes necessary.

In the following sections, we characterize the main steps of the ReGS execution.

3.1. Parameter Entry

Users submit jobs on an HTML page, that we call the *Entry Page*. The *Entry Page* is seen in Figure 2. Among the parameters shown, some are application parameters (for machine learning), while others are system-required.

After entering the parameters, the user will click a button on the lower part of the page. The request is sent to a machine in a Condor pool, which we call the *entry machine*. This machine contains Java servlets (gray boxes in Figure 1). One of them, the *Treatment Servlet*:

1. receives the parameters from the *Entry Page* and writes them into the *data file*;
2. calls the *Submitter*;
3. sends a message from the manager, Condor, back to the user, via another HTML page that we call the *Status Interface*. When the Submitter finishes, it passes the Condor message to the servlet.

The entry machine contains the whole structure of our tool, as described in Figure 1.

3.2. Directory structure creation and Job submission

Once it is called by the servlet, the *Submitter* begins its work. First, the *Directory structure* is created, using some of the input parameters. Within this *Directory structure*, data is automatically prepared for the experiments, which will spread a certain number of jobs. The data is internally represented using GRID-ADL, a script-like language that is parsed and converted into a graph of tasks. These tasks are clustered and submitted to the grid nodes. GRID-ADL is discussed into more detail in Section 3.4.

The *Submitter* creates the condor input source files for each job. In the machine learning experiment, these files contain Prolog queries [9]. It also creates the condor submission source file. Notice that the language used for the submission file can be any according to the resource manager that runs on the grid node.

Lastly, the *Submitter* submits the settled number of jobs to the Condor pool, with several machines. After this submission, the *Submitter* creates a MySQL database named `jobscondor`, with two tables: `pending` and `complete`. Each tuple in `pending` contains information about a job that has been submitted, but not yet finished. On the other hand, a tuple in `complete` contains information about a job that has already finished. From this information, we may retrieve the condor job ID (**jobid**), the date (**date**) and the time (**time**) of the job submission. For the `pending` table, there are two relevant fields: the path of the job's output file (**outfile**) and the number of the job's resubmissions until (**times-sub**).

When the *Submitter* finishes, it returns a Condor message to the *Treatment Servlet*, which by its turn repasses the message via the *Status Interface*. This interface (also a HTML

page) has two buttons for the user to click and thus know which jobs are still pending, and which are already complete, as shown in Figure 3.

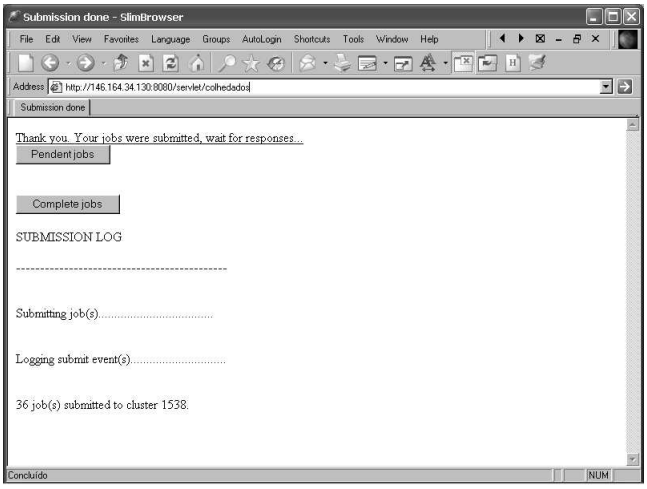

Figure 3. The *Status Interface*

3.3. Job monitoring

In order to monitor the newly-submitted jobs, a daemon, called the *Verifier*, is started. Its life cycle consists of:

- extract information from each tuple corresponding to a job in the `pending` table;
- verify if a job has finished and the integrity of its outputs;
- resubmit them or, in case of a successful finish, include a new tuple with information in table `complete`.

A job can be resubmitted in three cases:

1. If a job finishes without producing the expected output file;
2. If a job finishes, producing the expected output file, but this file is corrupted;
3. If a job exceeds the time limit in the Condor's job queue without producing the right output files.

We verify integrity of the output files by parsing them. If a syntax error occurs, the output is corrupted, and therefore, the job needs to be resubmitted. In the example application output files are Prolog files, and integrity is easily detected by loading the files to our Prolog system. Notice that ReGS must be informed on what is the output file format.

If a job needs to be resubmitted, a new tuple is created and included in the `pending` table, deleting the old one. Thus, the resubmitted job will have a new **jobid** and new **date** and **time** of submission. However, this new tuple uses values as **outfile** and **times-sub**, from the deleted tuple. Before being included into the new tuple, **times-sub** is incremented. We established **times-sub** to reach a maximum of *three* resubmissions, but this can be set by the user.

If the job finishes successfully and its output file is correct, a new tuple is created in `complete`, and its tuple in `pending` is deleted.

The Verifier daemon checks the jobs, resubmits them when necessary and updates the tables in `jobscondor`. Meanwhile, the *Status Interface* keeps the user updated with how jobs are being executed. Figure 3 presents an snapshot example.

A Java servlet, the *Pending jobs tracker*, is activated if a button on the *Status Interface* is pressed. This servlet will read each tuple in the `pending` table, and return its information (as **jobid**, **date** and **time**) to the user. The other button on the *Status Interface* will start a similar servlet, the *Complete jobs tracker*, which will read every tuple in the `complete` table, and return its information.

3.4. Description Language: GRID-ADL

ReGS uses GRID-ADL, a script-like language developed in the context of the GRAND middleware [11], to internally represent the user input data. GRID-ADL has a parser that converts the internal representation to a directed acyclic graph that can be conveniently partitioned to take advantage of the available grid nodes and their characteristics (such as bandwidth or data locality). The main advantage of GRID-ADL is that the user does not need to define the whole graph as the language provides iteration commands to the user. This becomes even more important during execution of the graph, because GRID-ADL does not need to represent all nodes in memory. Instead, graph creation is performed lazily.

Our language syntax is represented in Backus-Naur-Form (BNF) in Figure 4. Since it is self explanatory, we will not explain all details. We will only highlight the main aspects using the three DAG examples presented in Figure 5.

GRID-ADL can be considered as an extension of the DAGMan description language. Some main differences are the following:

- the user can give a hint on how the task graph can be classified ("independent", "loosely-coupled", or "highly-coupled"). This is useful to speedup the partitioning phase, since we use different algorithms for each type of graph;
- the task description presents, besides a name and a submission file, input and output file names;

```
<input_file> ::= [<comments>] [<graph_definition>]
                 <set_of_task_definition>
                 [<comments>]
                 [<transient_file_definition>]

<graph_definition> ::= "graph" <graph_type>

<graph_type> ::= "independent"
           | "loosely-coupled"
           | "high-coupled"

<set_of_task_definition> ::= <task_definition>
           | <loop>
           | <assignment>
           | <task_definition> <set_of_task_definition>
           | <loop> <set_of_task_definition>
           | <assignment> <set_of_task_definition>
           | <comments>

<task_definition> ::= "task" <task_name>
             "-i" <filenames> "-o" <filenames>
             [ "-c" <number> ]    [ "done" ]
<task_name> ::= <string> | <var>

<loop> ::= "foreach" <var> "in" <range> "{"
             <set_of_task_definition>   "}"
<range> ::= <number> .. <number>
          | "{" <symbols> "}"
<symbols> ::= <string>
          | <string> ";" <symbols>

<assignment> ::= <string> "=" <assignment'>
<assignment'> ::= <operator>
                | <operator> <operation> <operator>
                | ( <var>|<number> ) <math_operation>
                  ( <var>|<number> )
<operator> ::= <var> | <string> | <number>
<operation> ::= "+" | "-"
<math_operation> ::= "*" | "/" | "^"

<transient_file_definition> ::=
           "transient" <filenames>

<filenames> ::= <filename_unix>
             | <filename_windows>
             | <filename_unix> ";" <filenames>
             | <filename_windows> ";" <filenames>
<filename_unix> ::= <string>
             | <string> <filename_unix>
             | <string> "." <string>
             | "/" <string> <filename_unix>
<filename_windows> ::= <char> ":" <filename_windows2>
             | "\\" <string> "\" <filename_windows2>
<filename_windows2> ::= <string>
             | <string> "." <string>
             | "\" <string> <filename_windows2>

<var> ::= "${" <string> "}" <string> ::= <char>
             | <char> <any_char>
<any_char> ::= <char>
             | <special_char>
             | <digit>
             | <char> <any_char>
             | <special_char> <any_char>
             | <digit> <any_char>

<char> ::= a..z | A..Z

<special_char> ::= "_" | "-"

<number> ::= <digit>
         | <number> <digit>

<digit> ::= 0..9

<comments> ::= "#" <any_char> "\n"
```

Figure 4. Description Language Syntax in BNF

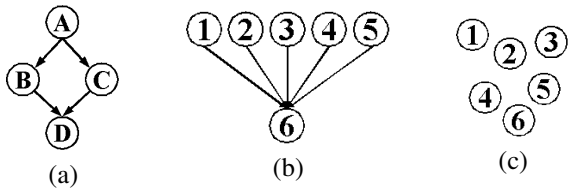

Figure 5. DAG used in input file examples

- some shell script like constructions are available to facilitate the description of tasks.

The first two differences can be seen in Figure 6, which describes the DAG of Figure 5(a). It presents an example where the user first indicates that his/her application has a loosely-coupled graph (first statement). The `graph` keyword is a hint the user gives to our system. It is an optional directive that should be used to get a better performance when trying to infer the DAG. For instance, if the user defines that the DAG represents independent tasks, it is not necessary to run the algorithm to infer the DAG since there is no precedence order between tasks.

The next four subsequent lines describe four tasks using the statement `task`. For each one the user needs to define a name, a submission file, one or more input files, and one or more output files. In ReGS, all this information is taken from the Entry Page.

```
graph loosely-coupled
task A A.sub -i data.in -o a.out
task B B.sub -i a.out -o b.out
task C C.sub -i a.out -o c.out
task D D.sub -i b.out c.out -o data.out
```

Figure 6. Input file for the DAG example 5(a)

This example is simpler than the transformations and derivations used by the Chimera description language while being more powerful than the DAGMan specification language.

Besides these direct and simple statements, we added to our language some shell script like constructions as illustrated in Figure 7, which is related to Figure 5(b). The second statement presents a string variable assignment. Next, an iteration command (`foreach`) is used to declare five tasks as well as to store in the string variable (`OUTPUT`) the output file names. Then, the string variable is used to indicate the input file of task 6. The final command defines that the files stored in the string variable would not be copied back to the user workspace, i.e, they are transient or temporary files.

The third example, in Figure 8, related to the DAG of Figure 5(c), illustrates with only three lines how to define an arbitrary number of tasks. In this case, we define

```
graph loosely-coupled
OUTPUT = ""
foreach ${TASK} in 1..5 {
 task ${TASK} ${TASK}.sub -i ${TASK}.in
                          -o ${TASK}.out
 OUTPUT = ${OUTPUT} + ${TASK}.out + " "
}
task 6 6.sub -i ${OUTPUT} -o data.out
transient ${OUTPUT}
```

Figure 7. Input file for DAG example 5(b)

an independent graph (a bag-of-tasks application) with six tasks. This example shows a nice feature of our language not present in the specification languages of systems like Chimera or DAGMan.

```
graph independent
foreach ${TASK} in 1..6 {
 task ${TASK} ${TASK}.sub -i ${TASK}.in
                          -o ${TASK}.out
}
```

Figure 8. Input file for DAG example 5(c)

We wrote a parser using the Java Compiler Compiler (JavaCC) for our language. JavaCC is a parser generator for the Java language.

Our prototype has the following functionalities:

- the input file is parsed and lexical and syntactic errors are detected. This is the most basic functionality required;
- during the parsing, application information is stored as a set of objects. This set of objects stores all information related to the tasks that allows the application to be further executed;
- any occurrence of the foreach statement is "executed" during the parsing phase. For instance, this allows task creation inside the loop to be properly expanded.

After the graph is built, we use different algorithms to group the tasks into clusters (according to the graph type) in order to maximise the computation/communication ratio. In the experiments presented in this paper, the application has only two phases, so the partitioning is straightforward.

ReGS takes the user input data and converts them to this intermediate language. It is straightforward to transform this language in DAGMan statements. Therefore, we use Condor DAGMan so that job dependencies can be tracked automatically, without any user intervention. A version of our tool containing DAGMan was created, and a simple example is described in Section 4.

	l=4			l=5
Total of jobs	108	216	252	126
Finished jobs	108	216	252	123
0 resubmissions	106	212	251	97
1 resubmission	2	4	1	20
2 resubmissions	0	0	0	5
3 resubmissions	0	0	0	1
Dismissed jobs	0	0	0	3
Avg. job allocation time (s)	3075	5218	6538	19814
Avg. job completion time (s)	3249	5531	6755	22650

Table 1. Initial Experiments

4. Results

This section presents a first study of the performance of our tool, and how it depends on Condor when allocating jobs to machines. We experiment with a particularly memory and CPU intensive application, which can be difficult to manage even in a small pool. The target application is *Machine Learning* (ML). A complete analysis of a learning can demand the execution of a massive number of experiments, and spread a large number of jobs. Our ML runs are particulary interesting because each run can have very different memory and CPU requirements. We can estimate lower-bound at submission time, but tasks can take significantly longer, and can use up a system's memory.

In order to perform a detailed analysis, we used a relatively small configuration: a cluster with 8 Intel Pentium V machines, with 512 MBytes of memory, and 1 GHz clock, where a Condor pool was installed and configured. The version of Condor we used was 6.4.7. The operating system used was Red Hat 7.2. Jobs that fail over 3 times are dismissed.

4.1. Performance Measurements

We ran several variations of an ML application on our pool. The application was on learning of carcinogenic components' characteristics in rodents . Performance strongly depends on a parameter l referring to clause length, that is, the maximal number of features that are expected to affect rodents carcinogenesis. We measured the number of failed jobs, the average job allocation time, and the average job completion time (both in seconds). By *allocation time* of a job, we mean the elapsed time between its submission to Condor environment and the allocation of a machine to the job; by *completion time* of a job we mean the time elapsed between its submission and its completion.

The first series of experiments spread 108 jobs, the second spread 216 jobs and the third spread 252 jobs - all of them with a clause length of 4. Note that a job's execu-

tion time in our experiments increases exponentially with the clause length, as it is clearly seen in the fourth series, which uses a clause length of 5. This seried spread 126 jobs, but their average job allocation and job completion times were very high.

Notice that even the first three series of experiments with low execution times presented failed jobs. Those jobs were detected and resubmitted, without user intervention. There was no need to look over hundreds of results in order to know if any output was corrupted or non-existent. None of those failed jobs was resubmitted either because the queue's time limit was exceeded (which was set as 1 day, but can be given by the user) or because the resubmission threshold limit was reached.

The fourth series (clause length=5) shows a harder problem. A percentage of 20,6% of the jobs failed - and in most cases this could only be detected by checking their output. Full automation is not sufficient: 3 jobs (2,4%) exceed our resubmission threshold (3). In this case this indicates some of the tasks may be exceeding total system memory, and may require user intervention.

4.2. Condor DAGMan and scheduler evaluation

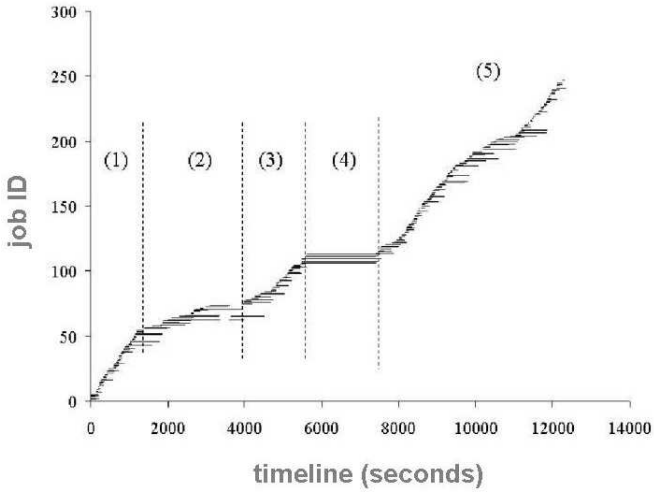

Figure 9. Graphic representation of all the jobs' executions in a series of experiments. A horizontal line is a job execution time (in seconds)

We rely on Condor DAGMan to detect dependencies among jobs. As the machine learning jobs tested are *loosely coupled*, we associated a "dummy job" (which produces a "Hello World" message) to each machine learning job. In other words, each "dummy job" is directly dependent on a machine learning job, in a 1:1 relation. A version of our tool constructs these dependencies automatically, by creating a DAGMan-syntax file and calling DAGMan after this. The user has only to inform about the executable program from which the dependent jobs will be created. During the tests, it was proved that all the "dummy jobs" only started after their parents finished.

We believe it would be interesting to evaluate the job scheduler system used in the series of experiments, so that we might be aware of limitations when scheduling jobs to the pool. To do so we use the third series of experiments, with 252 jobs. Figure 9 shows start and finish times for the sequence of tasks. .

The horizontal axis (x) in the graphic corresponds to the total execution time, in seconds, for the collection of 252 jobs. This time is elapsed from zero (the moment they were spread) to 12.327 (when the last job finishes), or 3 hours, 29 minutes and 50 seconds. The vertical axis (y) corresponds to the identification of the jobs, identified by numbers ranging from 0 to 252. The *execution time* of a job is an horizontal line, which for each job binds its allocation time to its completion time.

When a job is *checkpointed* and transferred to run in another machine, the execution time divides into two straight lines. The first one binds the point where the job is allocated and the point where it was *checkpointed*. The second one goes from the point where the job is allocated for the second time to the point where it completes. Of course, a job can be checkpointed more than once, but it didn't happen during this execution.

Taking these considerations into account, we present next a description of the relevant events that could be observed in Figure 9.

Interval	Allocation rate (job/no.seconds)
(1)	24,67
(2)	115,07
(3)	40,22
(4)	—
(5)	36,87

Table 2. Job allocation rates per interval

On the first 1.382 seconds, represented by interval (1), Condor could allocate one job each 24,67 seconds. Regarding to interval (2), there was a significant fall in this rate (one job each 115,07 seconds), with an increase of jobs' execution times. In interval (3), the rate increases again, reaching a value of one job allocation each 40,22 seconds.

A curious phenomenon is noticed in interval (4). Within this interval, no job allocation was done, and the jobs already allocated had a excessively long execution time. It was noticed that each of those jobs left the machine where it was, and Condor allocated it to another machine, in "jumps" without any sign of a checkpoint. The "jumps" are not reported inside the Condor log file, a file it creates for reporting several events from the jobs' executions.

On the last interval (5), the job allocation rate stabilized again, reaching a value similar to that from (1) - one allocated job each 36,87 seconds.

From this study, we conclude that the Condor scheduler can exhibit irregularities in making sure that a job, once allocated, executes without any interruption; it is known that, if a machine becomes suddenly busy (p.ex, a user opening a terminal), this can delay the job's execution - the job could be checkpointed or could stop and resume its execution in the *same* machine later. However, the irregularities were present even in the idle machines - and, during the interval (4), even the entry machine was not busy.

Incidents like those may be harmful to the experiments performance. A job similar to the ones we submitted, with a clause length of 4, can take about 3 minutes to complete. If we execute the 252 jobs within an uniprocessor environment, we shall obtain the total time of: 252*180 = 45.360 seconds.

On the other hand, the total time for executing the 252 jobs, with a peak of 7 machines, was 12.327 seconds. Therefore, the *speedup* obtained by the Condor is: 45.360/12.327 = 3,67. This result shows that the benefits of a multiprocessor environment might not be well used, which may justify a choice of the job scheduler by the user, depending on the situation. In our case, we were more concerned with throughput than with speedup, but in many cases speedup may be more important.

5. Conclusions and Future Work

We presented ReGS, a tool to enable the execution of a large quantity of experiments in a grid environment. ReGS is based on the ideas originally proposed by Dutra *et al* [2], and includes the following contributions: integration of all system components in a flexible tool, and automatic job management, including automatic error recovery. Our results show a real occurrence of errors, even for applications with a moderate number of jobs. We also observed that the number of errors tends to increase as we increase the number of jobs. We believe that ReGS can deal with this problem effectively.

Our first implementation of ReGS is geared towards machine learning applications. We are working on using ReGS to support other applications, namely in areas such as bioinformatics and physics.

So far, ReGS has relied on pure syntactic analysis to verify output file integrity. We are working on an interface that would allow users to introduce other rules, such as, file length, CRCs, or semantic-based rules.

We plan to allow ReGS to interface with different schedulers. Our experience has shown that the best job scheduler may vary with the application requirements. We believe that ReGS can be an effective tool to collect statistics that will help us in figuring out how efficient a job scheduler can be to run a given application.

It is also important to improve the interface with tools such as Condor DAGMan. Ideally, we would like to interface ReGS with GRAND [11], a middleware that employs a hierarchical archiecture to manage and control individual job submissions in grid environments.

References

[1] Platform Computing. `http://www.platform.com/products/overview.html`.

[2] I. Dutra, D. Page, V. Santos Costa, J. Shavlik, and M. Waddell. Toward automatic management of embarrassingly parallel applications. *Proceedings of International Conference on Parallel and Distributed Computing (Euro-Par)*, 2003.

[3] Sun Grid Engine. `http://www.sun.com/software/gridware/`.

[4] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*, chapter Computational Grids, pages 15–51. Morgan Kaufmann Pub., Inc., 1999.

[5] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *International Journal of Supercomputer Applications*, 15(3):472–602, 2001.

[6] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *14th Conference on Scientific and Statistical Database Management*, 2002.

[7] Y. Gil, E. Deelman, C. Kesselman, and H. Tangmurarunkit. Artificial intelligence and grids workflow planning and beyond. *IEEE Intelligent Systems*, January 2004.

[8] Portable Batch System. `http://www.openpbs.org`.

[9] The Yap Prolog System. `http://www.ncc.up.pt/~vsc/Yap/`.

[10] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley, 2003.

[11] P. K. Vargas, I. C. Dutra, and C. F.R. Geyer. Application partitioning and hierarchical management in grid environments. In *1st International Middleware Doctoral Symposium 2004*, pages 314–318, Toronto - Canadá, October 19th 2004.