Toward Automatic Management of Embarrassingly Parallel Applications

I. Dutra[1], D. Page[2], V. Santos Costa[1], J. Shavlik[2] and M. Waddell[2]

[1] Dep of Systems Engineering and Computer Science
Federal University of Rio de Janeiro, Rio de Janeiro, RJ, Brazil
{ines,vitor}@cos.ufrj.br
[2] Dep of Biostatistics and Medical Informatics,
University of Wisconsin-Madison, USA
{page,shavlik,mwaddell}@biostat.wisc.edu

Abstract. Large-scale applications that require executing very large numbers of tasks are only feasible through parallelism. In this work we present a system that automatically handles large numbers of experiments and data in the context of machine learning. Our system controls all experiments, including re-submission of failed jobs and relies on available resource managers to spawn jobs through pools of machines. Our results show that we can manage a very large number of experiments, using a reasonable amount of idle CPU cycles, with very little user intervention.

1 Introduction

Large-scale applications may require executing very large numbers of tasks, say, thousands or even hundreds of thousands of experiments. These applications are only feasible through parallelism and are nowadays often executed in clusters of workstations or in the Grid. Unfortunately, running these applications in an unreliable environment can be a complex problem. The several phases of computation in the application must be sequenced correctly: dependencies, usually arising through data written to and read from files, must be respected. Results will be grouped together, a summarised report over the whole computation should be made available. Errors, both from the application itself and from the environment, must be handled correctly. One must check whether experiments terminated successfully, and verify integrity of the output.

Most available software for monitoring applications in parallel and distributed environments, and more recently, in grid environments, concentrate on modelling and analysing hardware and software performance [8], prediction of lost cycles [9] or visualisation of parallel execution [12], to mention some. Most of them focus on parallelised applications. Few efforts have been spent on managing huge number of independent experiments and the increasing growth of interdisciplinary databases such as the ones used in biological or biomedical applications. Only recently, we have seen work in the context of the Grid such as the GriPhyN project for Physics [1], and the development of the general purpose system Chimera [6].

In this work we present a system originally designed to support the very large numbers of experiments and data in machine learning applications. We are interested in machine learning toward data mining of relational data from domains such as biochemistry [10, 4], and security [13]. Machine learning tasks are often computationally intensive. For instance, many learning systems, such as the ones that generate decision trees and logical clauses, must explore a so-called "search space" in order to find models that characterise well a set of correct (and possibly incorrect) examples. The size of the search space usually grows exponentially with the size of the problem. A single run of the learning algorithm can thus easily take hours over real data. Moreover, in order to study the effectiveness of the system one often needs to repeat experiments on different data. Splitting the original examples into different subsets or $f$olds and learning on each fold is also common. Thus, a single learning task may involve several independent coarse-grained tasks, providing an excellent source of parallelism. In fact, parallelism is often the only way one can actually perform such experiments in reasonable time.

Our system has successfully run several large-scale experiments. Our goal is to extract maximum advantage of the huge parallelism available in machine learning applications. The system supports non-interactive error-handling, including re-submission of failed jobs, while allowing users to control the general performance of the application. Whenever possible, we rely on available technology: for example, we use the Condor resource manager [2] to spawn jobs through pools of machines. We have been able to run very large applications. One example included over 50 thousand experiments: in this case we consumed about 53,000 hours of CPU, but the system took only 3 months to terminate, achieving peak parallelism of 400 simultaneous machines at a time, and requiring very little user intervention.

The paper is organised as follows. First, we present in more detail the machine learning environment and its requirements. We then discuss the motivation for an automatic tool. In section 2 we present the architecture of our system and the methodology applied to the two machine learning phases: experimentation and evaluation. We then discuss some performance figures and possibilities of enhancements. Last, we offer our conclusions and suggest future work.

2 A Tool for Managing Large Numbers of Experiments

In order to be able to run the many possible combinations of experiments one needs to perform to have statistically meaningful results in machine learning applications, in a feasible time, and handle the results with least possible user intervention to avoid manual errors, we developed a tool for job management of learning applications, currently supporting Linux and Solaris environments. This tool makes use of available resource manager systems to manage idle resources available. We address the following issues: *Data Management*: each experiment will have its own output and temporary files, but several experiments share input files. The system needs to create a directory structure such that the output and temporary files for individual experiments can be kept on separate, but easily accessed, directories for each experiment. *Control*: given a problem description our tool creates a set of build files to launch the actual jobs. Each script inputs the data required for a specific experiment, and sets the files that will be output. *Task Supervision*: Our tool must allow one to inspect successful and unsuccessful job termination. As discussed in more detail next, the probability that some jobs will fail is extremely high, so most cases of unsuccessful termination should be handled within the system. *User Interface*: a large number of results must be collected and displayed. Throughout, the user should be able to understand what is the current status of the computation, and plot the results.

In the experiments presented in this paper, we used as a resource manager, the Condor system, developed at the Computer Sciences Department of the University of Wisconsin-Madison. Condor is a specialised workload management system for compute-intensive jobs. Like other full-featured batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Among other things, Condor allows transparent migration of jobs from overloaded machines to idle machines and checkpointing, which permits that jobs can restart in another machine without the need to start from the beginning. These are typical tasks of a resource manager.

Putting it all Together Our architecture is composed of two main modules: (1) the Experimentation Module and (2) the Evaluation Module. The Experimentation Module is responsible for setting up the environment to start running experiments for one or several applications, for launching jobs for `tuning` and `cross-validation`, and checking if results are terminated and complete. Once all experiments have terminated, the Evaluation Module will consult the tuning results, if any, compute accuracy functions, and plot the results. If some job leaves the queue, but the output result is incorrect or corrupted, the `check termination` program will re-submit the job either to `tuning` or to `cross-validation`.

We provide a user-friendly web interface in to the system with options for choosing different learners and learning techniques. At the moment, our system deals only with Inductive Logic Programming [11] learners, but this interface is easily extensible to other learners.

Not Everything is Neat Several problems may arise while processing thousands or hundreds of thousands of experiments. We classify these problems as either *hardware-dependent* or *software-dependent*.

As we use off-the-shelf technology to launch our jobs, we expect such technology will deal correctly with hardware-dependent problems. In fact, the Condor system we use in our experiments deals with dynamic hardware changes such as inclusion of a new machine in the network or a machine or network failure. In the case of a new machine being included in the network, Condor updates the central manager table to mirror this change. In the case of a machine fault, Condor employs a checkpointing mechanism to migrate the job to another "healthy" machine.

With respect to software-dependent problems, we can enumerate several sources of possible failure: jobs get lost because a daemon stops working for some reason, a job breaks because of lack of physical memory, bugs in the machine learning system, bugs in the resource manager, or even bugs at the operating system level, corrupted data due to network contention, or lack of disk space.

None of the systems that we use is totally stable: during our experiments, it has happened that the operating system was upgraded while we were performing our experiments, and the task management software was incompatible with the new upgrade. Unforeseen situations, common in large software development, can lead the execution to crash. Problems can arise from any one of these components: the machine learning system, in some cases the software that runs the machine learning system, and the resource manager. As we rely in off-the-shelf components to build our system, some of these problems will be outside our control.

We have found memory management to be a critical issue for our application. The machine learning system that we use relies on dynamic memory allocation: thus, we do not know beforehand how much memory we use, and we can expect memory requirements to grow linearly with execution time. In general, we set the experiments to only run in machines with a minimal amount of memory, say M. If we set M too low, many experiments will fail. If we set M too high, we will severely restrict parallelism. Moreover, as machines with lots of memory are likely to be more recent and powerful machines, we can expect them to be busier.

In practice we can be sure that *runs will fail*. Our approach to deal with these problems is to have a daemon that inspects the job queues, and the application output files. So, if the daemon detects that some output file is not yet generated, and the job responsible for that output is no longer in the queue, it will re-submit the job. If some output file is not yet generated and the job is taking too long to produce an output, the daemon will remove the job from the queue and re-submit the job. An output file can be generated, but be corrupted. In that case, we also need to check for output syntax to be sure the next phase will collect correct data.

Note that these steps do not need any human intervention,[3] and do not require any change to the available software being used.

3 Performance Results

We ran three sets of experiments in a period of 6 months. The experiments concerned three relational learning tasks using Inductive Logic Programming techniques. The tasks included two biocomputing problems, one concerning the

[3] Of course, if the failure rate becomes excessive, the user is informed and is allowed to terminate the experiments.

prediction of carcinogenicity tests on rodents, and the second concerning a characterisation of genes involved in protein metabolism by means of their correlation with other genes. The third task used data on a set of smuggling *events*. The goal was to detect whether two events were somehow related.

We ran our jobs in the Condor pools available at the Biostatistics and Medical Informatics and the Computer Sciences Departments of the University of Wisconsin-Madison. We used PC machines running Linux and SUN machines running Solaris. Throughout, Condor collects statistics about the progress of the jobs, but as this consumes a large amount of disk storage, not all Condor pools keep all data about jobs. Therefore in Table 1 we show statistics only for one of our pools that stores the data information about jobs progress in disk. The first column shows the month when the experiments were running. The second column shows the total number of CPU hours spent by our jobs. The third column shows the average number of jobs running, followed by the average of jobs idle, and the peak of jobs running and idle. We can observe that the system has two activity peaks in June/July and in September. During September in average the tool was able to keep almost one hundred processors busy, just in this pool. The maximum number of jobs running in this pool was around 400.

The execution priority of jobs in Condor depends on the configuration and on the pool. If a job migrates to a remote pool of machines, the priority of this job may be lowered to guarantee that local users in the remote pool are not interfered by the "foreign" jobs. The statistics shown in Table 1 are related to a remote pool.

Table 1. Statistics of Jobs

Period	Tot Alloc Time (Hours)	JobsRunning Average	JobsIdle Average	JobsRunning Peak	JobsIdle Peak
May	639.6	12.8	337.6	61.0	1057.0
Jun	39375.5	67.3	5322.8	400.0	15312.0
Jul	18406.2	70.1	2243.9	207.0	13366.0
Aug	110.1	14.9	0.1	17.0	4.0
Sep	18283.8	93.5	6713.9	397.0	11754.0
Oct	1185.8	47.0	4027.3	122.0	5933.0

In order to better understand the table, Figure 1 shows the state of the above mentioned pool during the most active month, September. The X-axis corresponds to days of the week, and the Y-axis corresponds to total machines available in the pool. The red area (bottom colour in the figure) shows user-controlled machines, the blue area in the middle, idle machines, and the green area above shows machines running Condor tasks. Pool size varies between 700 and 900 machines, most of them Linux and Solaris machines. User activity is higher during week days, where users are working on their own machine. Condor most often keeps around 400 to 500 machines busy at any one time. This provides an insight into the maximum amount of parallelism we can take advantage of. Comparing with the results in Table 1 we can observe that there are points in time where we could actually take advantage of almost the full pool, even if the pool was shared with the whole campus.

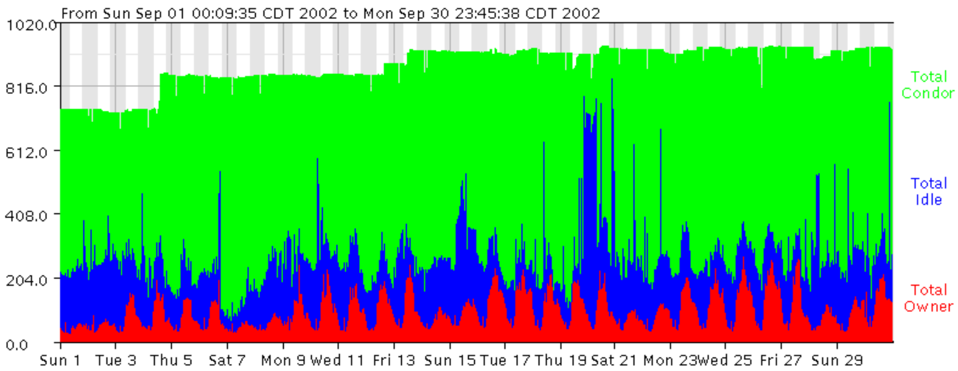

Fig. 1. The Condor Pool

Figure 2 shows that even during week days the system can provide an average of about 200 processors for long periods of time. This is about half of what we could expect from the cluster. Although we have little control over the cluster and we may be preempted by local tasks, even at the worst moments the pool can achieve above 50 machines simultaneously running our jobs.

Last, error handling was a very important issue in practice. From the 45,000 experiments we launched (three lots of 15,000 experiments for each application), around 20% failed for several reasons, and had to be re-submitted. This was done without user intervention.

4 Conclusions and Future Work

We described an automatic tool to manage large amounts of experiments in the machine learning context. The main advantage of such a tool is to provide a high-level user interface that can hide details of embarrassingly parallelisation, and that can allow for more automatic management of user experiments. Our system is capable of launching jobs automatically, check their integrity and termination, re-submit corrupted jobs, evaluate the results, plot relevant data, and inform the user about location of data and graphs. A second advantage of our tool is that we integrate off-the-shelf components in order to take advantage of already popular technologies. We used our tool to run several learning experiments with multirelational data. As an example, one of our experiments consumed about 53,000 hours of CPU, using a peak of 400 machines simultaneously.

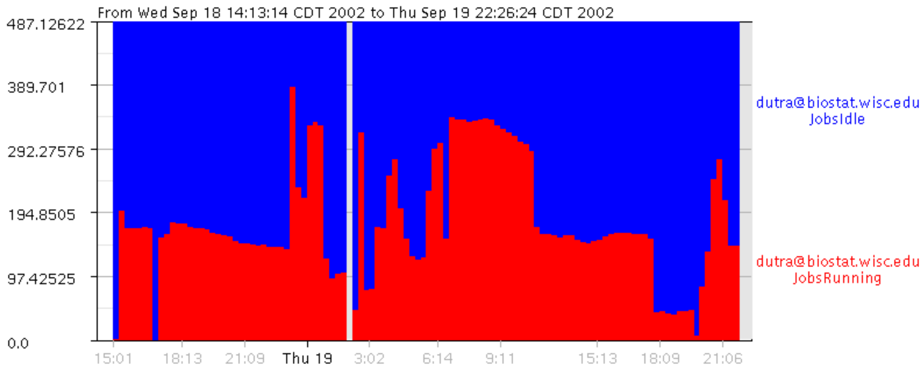

Fig. 2. Condor Activity

Most software available for parallel and distributed environments, and more recently, for grid environments, is designed to monitor applications by modelling and performance analysis, rather than managing a huge number of experiments. Condor has a limited form of handling experiments by allowing the user to express dependencies through the DAGMan tool [7]. But this tool requires that the user has some knowledge of ClassAds [14], a specification language for jobs and resources, in order to express dependencies explicitly. Chimera [6], one of the components of the Globus project [5], also has forms of automatically launching jobs depending on output produced by other jobs. Contrary to DAGMan, Chimera automatically generates a dependency execution graph, based on VDL (Virtual Data Language) [3] specifications.

Our proposal is to require minimum interference from the user, especially because many of the users in computational biology, where machine learning systems are heavily applied, have none or shallow knowledge of how to use a programming language.

As future work we intend to extend the tool to support other learning algorithms, such as the ones supported by the WEKA toolkit [15], and parallel boosting. We have been working on integrating our system with Chimera, in order to take advantage of one of the nicest features of Chimera that is to automatically infer dependencies between jobs in order to launch them without any user intervention.

Acknowledgments

This work was supported by DARPA EELD grant number F30602-01-2-0571, NSF Grant 9987841, NLM grant NLM 1 R01 LM07050-01, and CNPq. We would like to thank the Biomedical Group support staff and the Condor Team at the Computer Sciences Department for their invaluable help with Condor, Ashwin Srinivasan for his help with the Aleph system and the Carcinogenesis benchmark, and Yong Zhao from University of Chicago who has been helping us to integrate Chimera to our system.

References

1. The GriPhyN Project. White paper available at http://www.griphyn.org/ documents/white_paper/index.php, 2000.
2. J. Basney and M. Livny. Managing network resources in Condor. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylvania*, pages 298–299, Aug 2000.
3. A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.
4. I. Dutra, D. Page, V. Santos Costa, and J. Shavlik. An empirical evaluation of bagging in inductive logic programming. In *Proceedings of the Twelfth International Conference on Inductive Logic Programming*, Sydney, Australia, July 2002. Springer-Verlag.
5. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6), 2002.
6. I. Foster, J. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th conference on Scientific and Statistical Database Management (2002)*, 2002.
7. James Frey. Condor DAGMan: Handling Inter-Job Dependencies. http://www.cs.wisc.edu/condor/dagman/, 2002.
8. D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. NetLogger: A Toolkit for Distributed System Performance Analysis. In *Proceedings of the 8th International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '00)*, 2000.
9. W. Meira Jr., T. LeBlanc, and A. Poulos. Waiting time analysis and performance visualization in carnival. In *SPDT96: SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, PA, ACM*, pages 1–10, May 1996.
10. R. King, S. Muggleton, and M. Sternberg. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5:647–657, 1992.
11. N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications.* Artificial Intelligence. Ellis Horwood (Simon & Schuster), 1994.
12. B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.
13. R. Mooney, P. Melville, L. P. Rupert Tang, J. Shavlik, I. Dutra, D. Page, and V. Santos Costa. Relational data mining with inductive logic programming for link discovery. In *Proceedings of the National Science Foundation Workshop on Next Generation Data Mining*, Baltimore, Maryland, USA, 2002.
14. R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, Chicago, IL*, July 1998.
15. Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 1999.