Application Partitioning and Hierarchical Management in Grid Environments

Patrícia Kayser Vargas
UniLaSalle
Canoas, RS, Brazil
COPPE/Sistemas,
Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
kayser@cos.ufrj.br

Inês de Castro Dutra
COPPE/Sistemas,
Universidade Federal do Rio de Janeiro
Rio de Janeiro, RJ, Brazil
ines@cos.ufrj.br

Cláudio F. R. Geyer
Instituto de Informática,
Universidade Federal do Rio Grande do Sul
Porto Alegre, RS, Brazil
geyer@inf.ufrgs.br

ABSTRACT

Several works on grid computing have been proposed in the last years. However, most of them, including available software, can not deal properly with some issues related to control of applications that spread a very large number of tasks across the grid network. This work presents a step toward solving the problem of controlling such applications. We propose and discuss an architectural model called GRAND (Grid Robust ApplicatioN Deployment) based on partitioning and hierarchical submission and control of such applications. The main contribution of our model is to be able to control the execution of a huge number of distributed tasks while preserving data locality and reducing the load of the submit machines. We propose a taxonomy to classify application models to run in grid environments and partitioning methods. We also present our application description language GRID-ADL.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.11 [**Software Engineering**]: Software Architectures; D.3.2 [**Programming Languages**]: Language Classifications

General Terms

Design, Management

Keywords

grid computing, application management, scheduling

1. INTRODUCTION

A *grid computing* environment [4] is a distributed computing infrastructure that supports sharing and coordinated use of heterogeneous and geographically distributed resources. Resources can be CPUs, storage systems, or network interconnection as well as a special hardware, such as meteorological sensors or a specific software such as a database.

Several works on grid computing have been proposed in the last years. However, most work presented in the literature can not deal properly with some issues related to application management. Many applications have a high demand for computational resources such as CPU cycles and/or data storage. For instance, research in high energy physics (HEP) [2] and bioinformatics [8] usually requires processing of large amounts of data using processing intensive algorithms. Usually, these applications are composed of tasks and most systems deal with each individual task as if they are stand-alone applications. Very often, they are composed of hierarchical tasks that need to be dealt with altogether. These applications can also present a large-scale nature and spread a very large number of tasks requiring the execution of thousands or hundreds of thousands of experiments. Most current software systems fail to deal with two problems: (1) manage and control large numbers of tasks; and (2) regulate the submit machine load and network traffic. Dutra *et al.* [3] worked on item (1) and reported experiments of inductive logic programming that generated over 40 thousand jobs. However, this work was concentrated on providing an user level tool to control and monitor that specific application execution, including automatic resubmission of failed tasks.

Since applications that spread a large number of tasks must receive a special treatment for submission and execution control, we advocate that a hierarchy of managers that can dynamically distribute data and tasks can aid the application management. Our work provides a tool that can manage and control the two problems mentioned. It has a simple application description language as input. We consider that dependencies between tasks are programmed using data files. User describes only the task characteristics and the dependencies between tasks are inferred automatically through a data flow dependency analysis. Our architectural model handles two important issues in the context of applications that spread a huge number of tasks: (1) partitioning applications such that dependent tasks will be placed in the same grid node to avoid unnecessary migration of intermediate and/or transient data files, and (2) distributing the submission process such that the submit machine do not get overloaded. This text outlines our main contributions to these issues.

The remaining of this text is organized as follows. Section 2 presents some related works on grid computing. In Section 3 we present our application taxonomy. We present and discuss our architectural model in Section 4. Finally, Section 5 concludes this text.

2. GRID COMPUTING SYSTEMS

Using a timeline classification, Roure *et al* [13] identifies three generations in the evolution of grid systems. The first one includes the forerunners of grid computing. The second one have a focus on middleware to support large scale data and computation. The third one has an emphasis on distributed global collaboration, using a service oriented approach, and information layer issues.

Another very popular classification is based on the grid functionality. In that category we have two classes: computational or data grid. *Computational grid* focuses on reducing execution time of applications that require a great number of computer processing cycles. *Data grid* provides the way to solve large scale data management problems.

The two most popular grid-aware systems are Condor [15] and Globus [5]. Condor is a specialized 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. Globus, by its turn, is a whole framework that includes a set of services used, for example, to securely transfer files from one grid node to another (GridFTP), to manage meta data (MDS), and to allocate remote resources (GRAM). Condor-G [7] puts together Condor facilities to manage jobs with the Globus grid facilities such as secure authentication (GSI) and data transfer. Condor applies an opportunistic scheduling policy that concentrates on allocating idle resources to take advantage of idle CPU cycles. Globus focuses on providing several different services to execute secure code on authorized machines. Application management and control, load balancing and data locality are not their main focus.

Our work differs from those in the literature in two aspects. First, we handle three problems not approached by previous works: automatic application partitioning (taking into account data locality), automatic application management (when the application launches thousands of tasks that are not easily monitored by hand), and automatic control of the load in the submit machine. To the best of our knowledge, no other work in the literature solves these problems. Second, we do not propose a full grid solution (as the Globus project does, for example): we assume that issues like authorization and certification are provided to grant access to the grid nodes, as well as we assume each grid node has its own local resource manager.

3. APPLICATION TAXONOMY

As mentioned in section 1, we consider applications composed by many tasks that can have dependencies only through file sharing. Applications can be modeled as a dependency *graph* of tasks due to file sharing. For example, if a task a produces an output file f_a that task b uses as its input file, then b must wait until task a finishes. In this example, a and b are nodes and there is an edge from a to b. We consider that the application graph is a directed acyclic graph ((DAG). This is a common assumption as presented in Condor's DAGMan [15] and Globus' Chimera [6].

We classify those applications in three types:

Independent tasks: is the simplest kind of application, usually called "bag-of-tasks". In these applications all tasks are independent. Normally, data generated by each task are considered part of the final result but they do not need to be processed by any specific process. One example that falls in this category are Monte Carlo simulations typically used in HEP and engineering experiments [18]. MyGrid [10] is a grid system that deals properly with this kind of application.

Loosely-coupled tasks: has very few dependencies and thus has a graph with few sharing points. It is typically characterized by an application divided in phases. For example, the application distributes data in the beginning of the computation to several processes and collects them after every process finishes to submit to an analysis task to generate the final result. One example is the experiment mentioned in Dutra *et al.* [3].

Tightly-coupled tasks: has highly complex graphs. This kind of application is not so common and is more difficult to be partitioned. One example that falls in this category is Finite Constraint Satisfaction Problems, where each task is responsible for solving one optimization sub-problem that shares variables with other sub-problems.

4. TOWARD A HIERARCHICAL APPLICATION MANAGEMENT SYSTEM

We designed an architectural model to be implemented at a middleware level. In order to design our architecture, we consider that the resources and tasks are modeled as graphs. At the resource side, each node corresponds to a grid site, and therefore has information about individual resources in the site, including access restrictions, computational power, costs etc. The edges of the resources graph correspond to the connections available between the grid nodes. At the application side, each node corresponds to an application task, and each edge represents a task dependence that indicates a precedence order. This architectural model relies on already available software components to solve issues such as allocation of tasks within a grid node, or authorization control.

The main premises assumed to the conception of our model are the following:

- A huge number of tasks can be submitted.
- In our applications, tasks do not communicate by message passing.
- Tasks can have dependencies with other tasks due to file sharing.

- Huge number of files can be manipulated.
- Underlying grid environment is secure.
- Each grid node has its local Resource Management System (RMS).

We also assume that once a task is allocated it will not be scheduled again or at least this will be transparent to the higher level resource managers.

4.1 Describing the application

Generally, users run their jobs using some kind of description file that contains characteristics such as the task to be executed, the computational power required or the full path to the executable. As most users are acquainted with this kind of routine, we opted to maintain this classical approach, and provide to the user a description language that can quickly represent the user applications and needs. Our description language is called GRID-ADL (Grid Application Description Language) and has the legibility and simplicity of shell scripts and DAGMan [15] language while presents data relations that allow to automatically infer the DAG in the same way Chimera [6] does. The user submits a file describing only the kind of application (independent, loosely-coupled, or tightly-coupled tasks) and its tasks indicating input and output files.

Figure 1 shows one example of DAG and Figure 2 presents its corresponding input description file. This example describes a loosely-coupled graph as the first line indicates with the `graph` keyword. This keyword is a hint the user gives. It is a 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.

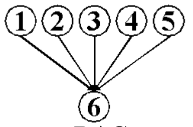

Figure 1: DAG example

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

Figure 2: Input description file example

The example represents an application composed of 6 tasks, where five of them can proceed in parallel and one of them (called 6) waits for the results of the five preceding computations to accomplish its final result. In this case, each task is defined using the keyword `task` followed by its logical name, the associated executable, and its input and output files (respectively `-i` and `-o`) as we can see in lines 4 and 7.

Notice that this example presents some important features of our description language: the use of a well know syntax and common commands that simplify coding and reading. In this example, a string variable `OUTPUT` is created initially empty in line 2, then the output file names of the first five tasks are concatenated in line 5. This simplifies the input file description of task 6 (line 7) and the description of the `transient` files (line 8). The keyword `transient` is used to say that the specified files do not need to be transferred back to the user local file system. Notice that the example files only can be removed after task 6 finishes its execution. Finally, our language has an iterator command `foreach` to make easier the description of a large number of tasks.

Once the system generates the graph, partitioning is performed using different algorithms depending on the kind of graph. If the application is of the kind **independent tasks**, each task can be assigned to any grid node or run locally subject to scheduling constraints (e.g., task requirements, link capacity and associated grid node cost). If the task is of the kind **loosely-coupled**, a depth-first search algorithm can be used to partition the graph in subtrees. If the task is of the kind **tightly-coupled** a more complex partitioning algorithm is required. We have been working on algorithms based on the Scheduling by Edge Reversal technique [1, 11].

4.2 Submitting the Application

Our proposal for application submission is a management mechanism called GRAND (Grid Robust ApplicatioN Deployment) [16, 17]. The GRAND model can be classified as a high-level scheduler [14] since it queries other schedulers for possible allocations. It can also be considered as a Metascheduler [12] because it allows to request resources of more than one machine for a single job.

The application submission and control is done through the following hierarchy of managers: (level 0) the user submits an application in a submit machine through the `Application Manager`; (level 1) the `Application Manager` partition the application in subgraphs and send to some `Submission Managers` the task descriptions; (level 2) an `Submission Manager` instantiates on demand the `Task Managers` that will control the task submission to the local RMSs on the grid nodes; (level 3) RMSs on grid nodes receive requests from our `Task Managers` and schedule the tasks to be executed locally.

The higher level of the application control must infer the DAG and make the partitioning. The `Application Manager` is in charge of: (1) processing the user submit file describing the tasks to be executed; (2) partitioning the tasks into subgraphs; and (3) showing, in a user friendly way, the status and monitoring information about the application.

Subgraphs defined by the partitioning algorithm are assigned to the second level controllers, called `Submission Managers`, which will instantiate the `Task Manager` processes to deal with the actual submission of the tasks to the nodes of the grid. This third level is necessary to isolate implementation details related to specific local resource managers. The `Submission Manager` main functions are :

- to create `Task Manager` to control actual task execution. Each daemon keeps control of a subgraph of tasks defined by the partitioning;
- to keep information about computational resources;
- to supply monitoring and status information useful to the user. It stores in log files the information in a synthetic way. This information is sent to the `Application Manager` that presents it to the user. This periodic information flow is also used to detect failures.

The `Task Manager` is responsible for communicating with remote machines and launching tasks to remote nodes. It works similarly to a wrapper being able to communicate with a specific local resource manager. For example, a `Task Manager` is instantiated to communicate with a grid node that uses PBS while another `Task Manager` can be instantiated to communicate with another grid node that uses SGE.

Figure 3 illustrates the three main components of our model and their relationship. We assume that our hierarchy of managers is running in the local network to (1) avoid forcing that other sites run our daemons, and (2) to minimize communication time between managers.

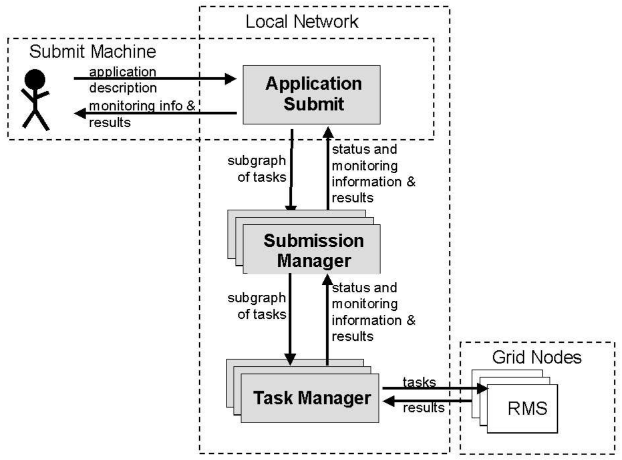

Figure 3: Hierarchical task management main components

When the user submits his/her application in the submit machine, the `Application Manager` can already be active or can be started due to the current request. When an `Application Manager` becomes active, it broadcasts a message to its local network. All `Submission Managers` reply to this message to inform their location and status. When an application submission request arrives to the active `Application Manager`, the application is partitioned in subgraphs. The `Application Manager` uses its local information and choose one or more `Submission Managers` to accomplish the tasks. The choice is made using the following criteria based on heuristics:

- the `Submission Managers` that have recently communicated with the `Application Manager` and reported that are not overloaded have preference to receive subgraphs. The periodical communication can detect when a `Submission Manager` is faulty or overloaded;
- the computational power of the machine, considering CPU and memory, determines the upper bound on the number of subgraphs a `Submission Manager` can receive. The more memory and CPU power a machine has, the more subgraphs its `Submission Manager` can handle. This aims at avoiding to overload the submit machine;
- the `Application Manager` keeps a weight for each `Submission Manager`. Greater values indicate powerful `Submission Managers`. This value is based on previous executions data and indicates how well the `Submission Manager` accomplished the tasks it received.

The chosen `Submission Managers` will receive subgraphs in an internal representation. At this moment, there is no transfer of executables or input files. Then, periodically the `Submission Managers` will communicate to the `Application Manager` the execution progress. This communication allows monitoring information to the user and also fault detection. Communication between the `Submission Managers` can happen, since some tasks in different subgraphs can have dependencies. Therefore some synchronization points must be established. The `Application Manager` must send, included in the subgraph description, the identification of each manager that is related to this subgraph. For example, suppose a `Submission Manager` sm_2 has a task B which must be executed after task A assigned to `Submission Manager` sm_1. In this case, sm_1 must send a message to sm_2 when task A finishes.

Each `Submission Manager` must find the most suitable resources to run its subgraphs. A `Submission Manager` chooses a grid node using the following criteria:

- the `Submission Managers` keep a list of available grid nodes. Some subgraphs will have requirements that just some grid nodes can match. Thus, grid nodes must match tasks requirements to be selected;
- for each grid node, an upper bound on the number of subgraphs it can receive will be estimated. Besides, the ongoing submissions are taken into consideration;
- the `Submission Manager` keeps information about previous executions. It uses this information to calculate a weight based on the application characteristics. Greater values for the weight indicate “better” grid node candidates.

It is required a `Task Manager`, in the same machine of the `Submission Manager`, for each grid node a `Submission Manager` can access. `Task Managers` can be dynamically activated and deactivated according to the `Submission Manager` demands. The `Submission Manager` sends the subgraph to a `Task Manager` according to the grid node chosen. The `Task Manager` is responsible for translating the internal subgraph description to the appropriate format for tasks submission. For example, if it communicates with a Condor pool, it must prepare a Condor submit file and send the command to start tasks.

4.3 Implementation Issues

In a first approach, we decided for model simulation. Simulation has several advantages, the most important being that we can isolate the model from operational issues such as the presence of firewalls or access restrictions. Other advantages are that we can control several parameters, for example, number of nodes in a cluster, bandwidth, latency, and mean time between failures, and can reproduce experiments. From the simulation experiments we can extract important implementation decisions. We are using the Monarc 2 Simulator [9]. Monarc is a java discrete-event simulator which allows to model the main grid abstractions such as resources and tasks. We are also implementing a GRAND prototype using JavaCC and ISAM/EXEHDA system.

5. FINAL REMARKS

This paper presented a general framework for application management in grid environments, whose central idea is to have a hierarchical organization of controllers, where load of the user machine (submit) is shared with other machines. Our proposal wants to take advantage of hierarchical structures, because this seems to be the most appropriate organization for grid environments. Our architectural model handles two important issues in the context of applications that spread a huge number of tasks: (1) partitioning applications such that dependent tasks will be placed in the same grid node to avoid unnecessary migration of intermediate and/or transient data files, and (2) distributing the submission process such that the submit machine do not get overloaded. As far as the authors know, this is the first proposal of a hierarchical application management system for grid environments and is the first work that focuses on data locality to make scheduling decisions [16, 17].

We are now designing our simulation model. With simulation, several parameters will be under control and we will try to get the best options to our system implementation. As future work, we will finish the implementation of our model and test it using applications from engineering, through a collaboration with the Laboratório de Projeto de Circuitos at UFRJ, from high energy physics, through the HEPGrid collaboration, and from machine learning applied to biological data through a collaboration with the Department of Biostatistics and Medical Informatics at University of Wisconsin.

6. ACKNOWLEDGMENTS

The authors would like to thank Marluce Rodrigues Pereira and Vítor Santos Costa for their comments during the design of the GRAND model. This work is partially supported by the Brazilian Research Council CNPq.

7. REFERENCES

[1] V. C. BARBOSA and E. GAFNI. Concurrency in heavily loaded neighborhood-constrained systems. *ACM Transactions on Programming Languages and Systems*, 11(4):562–584, October 1989.

[2] The compact muon solenoid (CMS) project, 2003. `http://lcg.web.cern.ch/`.

[3] I. C. DUTRA, D. PAGE, V. SANTOS COSTA, J. SHAVLIK, and M. WADDELL. Toward automatic management of embarrassingly parallel applications. In *26th International Conference on Parallel and Distributed Computing (Europar 2003)*, pages 509–516, Klagenfurt, Austria, August 2003.

[4] I. FOSTER and C. KESSELMAN. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, California, USA, 1 edition, 1998.

[5] I. FOSTER, C. KESSELMAN, J. NICK, and S. TUECKE. Grid services for distributed system integration. *IEEE Computer*, 35(6), June 2002.

[6] I. FOSTER, J. VOECKLER, 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*, Edinburgh, Scotland, July 2002.

[7] J. FREY, T. TANNENBAUM, I. FOSTER, M. LIVNY, and S. TUECKE. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.

[8] K.-i. KURATA, C. SAGUEZ, G. DINE, H. NAKAMURA, and V. BRETON. Evaluation of unique sequences on the European data grid. In *Proceedings of the First Asia-Pacific bioinformatics conference on Bioinformatics 2003*, pages 43–52. Australian Computer Society, Inc., 2003.

[9] I. LEGRAND and H. B. NEWMAN. The MONARC toolset for simulating large network-distributed processing systems. In *Winter Simulation Conference 2000*, pages 1794–1801, 2000.

[10] D. S. PARANHOS, W. CIRNE, and F. V. BRASILEIRO. Trading cycles for information: Using replication to schedule bag-of-tasks applications on computational grids. In *Proceedings of the 26th International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, August 2003.

[11] M. R. PEREIRA, P. K. VARGAS, F. M. G. FRANÇA, M. C. S. CASTRO, and I. C. DUTRA. Applying Scheduling by Edge Reversal to constraint partitioning. In *The 15th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, São Paulo, SP, November 10-12 2003.

[12] M. ROEHRIG, W. ZIEGLER, and P. WIEDER. Grid scheduling dictionary of terms and keywords. Document gfd-i.11, Global Grid Forum, Nov 2002. Available at `http://forge.gridforum.org/projects/ggf-editor/document/GFD-I.11/en/1`.

[13] D. D. ROURE, M. A. BAKER, N. R. JENNINGS, and N. R. SHADBOLT. The evolution of the Grid. In F. BERMAN, G. FOX, and T. HEY, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 65–100. Wiley & Sons, 2003.

[14] U. SCHWIEGELSHOHN and R. YAHYAPOUR. Attributes for communication between scheduling instances, December 2001. Available at `http://ds.e-technik.uni-dortmund.de/~yahya/ggf-sched/WG/sched_attr/Sch%edWD.10.6.pdf`.

[15] 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.

[16] P. K. VARGAS, I. C. DUTRA, and C. F. GEYER. Hierarchical resource management and application control in grid environments. Technical Report ES-608/03, COPPE/Sistemas - UFRJ, 2003.

[17] P. K. VARGAS, I. C. DUTRA, and C. F. GEYER. Application partitioning and hierarchical application management in grid environments. Technical report, COPPE/Sistemas - UFRJ, 2004.

[18] A. YAMIN *et al.* A framework for exploiting adaptation in high heterogeneous distributed processing. In *Proceedings of the XIV Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Vitória, ES, October 28–30 2002.