SPHINX: A SCHEDULING MIDDLEWARE FOR DATA INTENSIVE APPLICATION ON A GRID

J. In, P. Avery, R. Cavanaugh, L. Chitnis, M. Kulkarni, S. Ranka,
University of Florida, Gainesville, FL, 32611 USA

Abstract

A grid consists of high-end computational, storage, and network resources that, while known a priori, are dynamic with respect to activity and availability. Efficient co-scheduling of requests to use grid resources must adapt to this dynamic environment while meeting administrative policies. We discusses the necessary requirements of such a scheduler and introduce a distributed framework called SPHINX that schedules complex, data intensive High Energy Physics and Data Mining applications in a grid environment, respecting local and global policies along with a specified level of quality of service. The SPHINX design allows for a number of functional modules and/or distributed services to flexibly schedule workflows representing multiple applications on grids. We present experimental results for SPHINX that effectively utilizes existing grid middleware such as monitoring and workflow management/execution systems. These results demonstrate that SPHINX can successfully schedule work across a large number of grid sites that are owned by multiple units in a virtual organization.

INTRODUCTION

The execution of user applications must simultaneously satisfy both job execution constraints and system usage policies. First, grid resources are geographically distributed and heterogeneous in nature. One of the central concepts of a grid is that of a Virtual Organization (VO) [1], which is a group of consumers and producers united in their secure use of distributed high-end computational resources towards a common goal. Actual organizations, distributed nationwide or worldwide, participate in one or more VOs by sharing some or all of their resources. Second, these grid resources have decentralized ownership and different local scheduling policies dependent on their VO. Third, the dynamic load and availability of the resources require mechanisms for discovering and characterizing their status continually [1]. Although the above referenced systems address one or more of these characteristics, they do not address all of them in a cohesive manner for grids.

REQUIREMENTS OF A GRID SCHEDULING INFRASTRUCTURE

To efficiently schedule jobs, a grid scheduling system must have access to important information about the grid environment and its dynamic usage. Additionally, the scheduling system must meet certain fault tolerance and customizability requirements.

Information Requirements

A core requirement for scheduling work across a dynamic grid environment is situational awareness. Indeed, several monitoring systems exist which provide, in principle, a large volume of both static and dynamic information. Here, we briefly identify some basic observables which directly impact placement decisions and indicate where important work remains in publishing such information.

<u>*Dynamic Grid Weather*</u>: It is our experience that, in the context of grid computing where a gatekeeper functions as a single point of entry to a locally managed cluster of worker-nodes, individual load averages on worker-nodes do not directly assist in good job placement choices. Of more direct importance is site-*aggregate* information such as the queue-depth (e.g. total number of waiting jobs, total number of running jobs, and total number of available slots) as reported by the local scheduler on the remote grid-site. Also directly important are the amount of storage space currently available (per storage volume) on the remote grid-site and the current round-trip-time between the grid-site and possible external data transfer points.

<u>*Static Policies and Resource Descriptions*</u>: Inter- and intra-VO policy information as well as resource property descriptions also directly impact scheduling decisions and tend to be more static in nature. Particularly useful to scheduling systems are grid-site usage policies which can be stated in terms of (or translated to) resource usage "quotas" and "relative priorities" on a particular grid-site. Further, while most grid-sites do provide publish, via grid-information systems, descriptions of their resource properties (including the operating system type, CPU type, etc), most currently fall short in describing their full execution environment (e.g. locally installed gcc and libc versions) which is often vital in order to satisfy application requirements.

System Requirements

While the kinds of information above should be available to the system for efficient grid scheduling, the following requirements must be satisfied to provide efficient scheduling services to a grid VO community.

<u>*Distributed, Fault-tolerant Scheduling*</u>: The need for fault tolerance gives rise to a need for a distributed scheduling system. Centralized scheduling leaves the grid system prone to a single point of failure. Distributing

the scheduling functionality between several agents is essential to providing the required fault tolerance.

Customizability: Within the grid, many different VOs will interact within the grid environment and each of these VOs will have different application requirements. The scheduling system must be customizable enough to allow each organization with the flexibility to optimize the system for their particular needs.

Extensibility: The architecture of the scheduling system should be extensible to allow for the inclusion of higher level modules into the framework. Higher level modules could help map domain specific queries onto more generic scheduling problems and map domain specific constraints onto generic scheduling constraints.

Interoperability with Other Scheduling Systems: Any single scheduling system is unlikely to provide a unique solution for all VOs. In order to allow cooperation at the level of VOs, for example in a hierarchy of VOs or among VO peers, the scheduling system within any single VO should be able to route jobs to or accept jobs from external VOs subject to policy and grid information constraints.

Quality of Service: Multiple qualities of service may be desirable as there are potentially different types of users. There are users who run small jobs that care about interactive behaviour from the underlying system. On the other hand, large production runs may be acceptably executed as batch jobs. Users may put deadlines by which submitted jobs should be completed. The scheduling system should be able to provide these differential QoS features for the administrator/users.

SPHINX

The scheduling system presented in this paper, SPHINX, works to incorporate the infrastructure requirements described above using a client-server model based on the Clarens Web Service Backbone [2]. Such a model provides flexibility in defining the precise distributed scheduling services architecture. This allows, for example on the one hand, many light-weight clients to exploit a few powerful servers or, on the other hand, many heavy-weight clients to function autonomously, by each running their own personal server. Figure 1 shows the current Sphinx system infrastructure.

The Client

The scheduling client is an agent for processing user scheduling and execution requests. It interacts with both the scheduling servers, which recommend resources for task execution, and a grid execution system (e.g. VDT and Condor-G/DAGMan [3]). Because of this close connection to external components, it is often important that the client be as light-weight and flexible as feasible. In this way, the client can be easily modified if external components change. A Message Interface allows controlled access to the Sphinx Data Warehouse enabling customized planning modules to also be located, in principle, on the Sphinx Client.

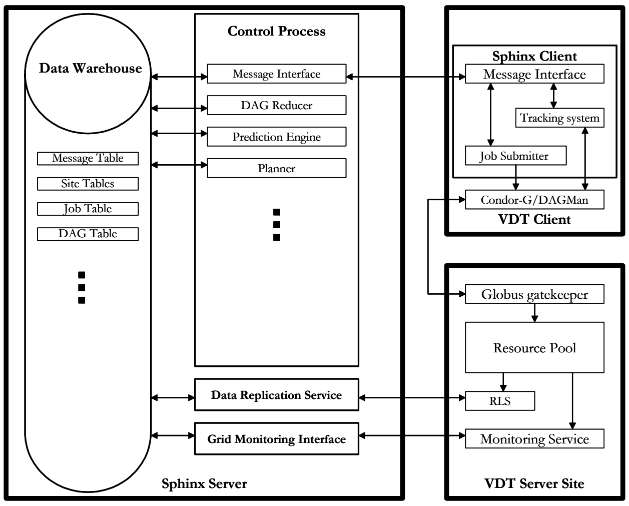

Figure 1: The Sphinx scheduling system infrastructure and its relation to the GriPhyN Virtual Data Tookit (VDT) [3].

The Server

The central components of the SPHINX system are the scheduling servers which use a database infrastructure to maintain the state of the current scheduling process. This not only simplifies development, but also provides a fault tolerant system for inter-process communication, and a robust recovery system that can detect internal component failure. The architecture also allows for the addition of new modules without necessarily affecting the logical structure of already written modules. These could be provenance tracking or other high level modules, for example. Currently, such planning modules are predetermined at runtime.

A control process enables the server to function as a finite state machine, calling the appropriate planning module to change the state of a DAG or job within a DAG. In addition, the database maintains several tables: a transformation catalogue cache, a replica catalogue cache, list of resources, and job-tracking information. Together these components move each DAG through six states (unreduced, unpredicted, unaccepted, unplanned, unfinished, remove) while they move each job in the DAG through five similar states (unpredicted, unaccepted, unplanned, unsent, remove).

The Scheduling Process

The user initially requests an abstract workflow (DAG file) be produced by a Workflow Planner (such as the Chimera Virtual Data System [4]) and then passes it to the scheduling client. This abstract plan describes the logical I/O dependencies and the executables within a group of jobs, but does not contain any location or data-existence information. After parsing the DAG, the client sends a message containing the DAG information to the scheduling server.

Upon receipt of the abstract DAG, the server processes that DAG and returns back to the client individual job submission requests from the server. Each of these messages contains an execution plan for a particular job within the submitted DAG. The client reads these plans and constructs the appropriate submission format to submit the job to the site. This provides the client with the freedom to formulate the actual job submission in terms of any number of existing job description languages.

After the job has been submitted, the client reports the success of the submission to the server. A job tracking module in the client keeps track of the status of submitted jobs to monitor their execution. If the execution is held or killed on remote sites, then the client reports the status change to the server requesting re-planning of the killed or held jobs. The client also sends the job cancellation message to the remote sides on which the held jobs are located.

POLICY BASED SCHEDULING AND QUALITY OF SERVICE

Grid computing requires collaborative resource sharing within a Virtual Organization (VO) and between different VOs. Resource providers and request submitters who participate within a VO share resources by defining how resource usage takes place in terms of where what, who, and when it is allowed. Accordingly, we assume that policies may be represented in a three dimensional space consisting of resource provider, request submitter, and time [5].

By exploiting the relational character of policy attributes, a policy description space is conveniently represented as a recursive, hierarchical tree. Indeed, the heterogeneity of the underlying systems, the difficulty of obtaining and maintaining global state information, and the complexity of the overall task all suggest a hierarchical, recursive approach to resource allocation. Further, such a hierarchical approach allows for a dynamic and flexible environment in which to administer policies.

Three of the dimensions in the policy space, consisting of resource provider, request submitter and time, are modelled as hierarchical categorical policy attributes expressed in terms of quotas. Administrators, resource providers or requesters who participate in a VO then define resource usage policies (in terms of quotas) involving various levels of this hierarchical space.

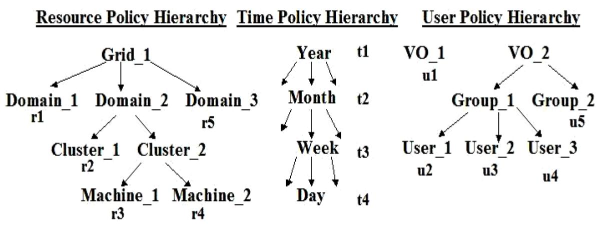

Figure 2: Policies and resource allocation are described in a hierarchical, recursive way.

Sphinx uses optimised constraint methods to decide the placement of the job [5]. Specifically, we apply a scalar objective function ***f(x)*** along with a policy matrix ***A*** which constrains the search for an optimal solution vector ***x*** for allocating resources in support of the requirements from a particular request vector ***b*** using the well known method of Linear Programming (LP): find ***x*** where $\boldsymbol{A\,x} \geq \boldsymbol{b}$ subject to **min(*f(x)*)**. The optimality of ***x*** depends on the algorithm ***f(x)*** and can be modified to suit the particular requirements of an individual user or VO.

In future releases, a request for a particular quality of service, such as a completion deadline, will be expressed as an additional requirement in ***b***. We anticipate that by employing statistical time series techniques and considering current grid weather, a probability for meeting the deadline may be estimated. Before job submission, the Sphinx Server would provide a Quality of Service statement (i.e. a probability to meet the specified deadline) to the Sphinx Client, whereupon the client may decide to proceed or to modify the workflow to more probably fit within the client's self defined time constraint.

EXPERIMENTAL RESULTS

While the SPHINX framework continues to be developed, several important milestones have been already reached. We report on one recent experiment result in which the SPHINX framework used Grid3 resources [6].

Two competing SPHINX servers were setup to execute identical workflows simultaneously: (1) one which did not make use of grid weather from Grid3 monitoring systems, and (2) one which made explicit use of grid weather from Grid3 monitoring systems. Both SPHINX servers used the LP based scheduling method described in the previous section, however, the particular algorithm employed differed between the two servers. The server which did not use monitoring information employed a simple round-robin load balancing algorithm, whilst the server which did use monitoring information employed a time dependent objective function, accounting for the total number of jobs idle and running on each Grid3 Computing Element (CE) gatekeeper.

Both servers were given identical, but separate workflows consisting of 30 Workflow DAGS, each with 10 jobs. For each job, the SPHINX client created a "job-DAG" consisting of several sub-jobs which: (1) create the job environment on the remote grid-site, (2) transfer input data to the remote grid-site, (3) execute the job on the remote grid-site, (4) transfer output data to a 3rd party site, (5) publish the location of the output data, (6) clean up the job environment on the remote grid-site. In total, accounting for all jobs and sub-jobs, both servers managed 1800 submissions each, from their respective SPHINX Clients (30 workflow DAGs, 10 "job-DAGs" per workflow DAG, and 6 sub-jobs per "job-DAG").

The conditions of Grid3 at the time of the experiment was that of a moderately loaded grid; ATLAS Data Challenge 2 production and other iVDGL work where being conducting on Grid3 at the same time that this experiment was conducted. Tight stability "cuts" where imposed on each grid-site. For example, if the execution time for a job was significantly longer than the *a priori*

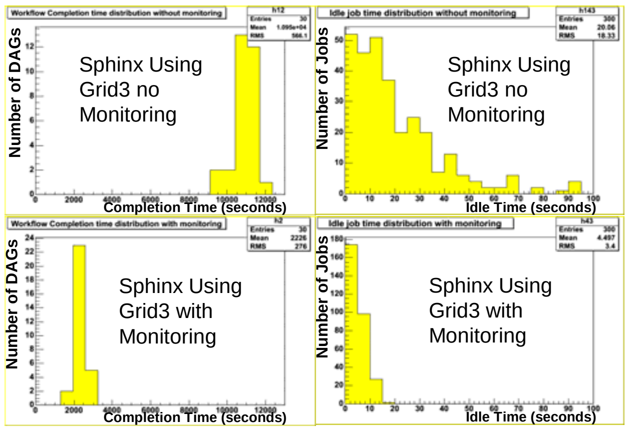

Figure 3: Experimental Results on Grid3. **Top left**: DAG completion time when no grid weather was used for scheduling. **Top right**: Job idle time spent in the remote queue when grid weather was not used in job placement decisions. **Bottom left**: DAG completion time when grid weather was used in the scheduling process. **Bottom right**: Job idle time spent in the remote queue in which grid weather was used in job placement decisions.

user stated “required” execution time, the job was automatically killed, submitted to a different grid-site, and the original grid-site removed from a list of “good” grid-sites. Once a steady state of “good” grid sites was achieved, 13 out of 27 original Grid3 sites were used for the experiment.

Figure 3 shows the results of the experiment. The two histograms across this top row represent the workflow completion time and job idle time, respectively, for the SPHINX server which did not use grid weather. The two histograms across the bottom row represent the workflow completion time and job idle time, respectively, for the SPHINX server which did use grid weather information. In particular, one readily sees (in the two left-hand plots) that the server which made use of monitoring information outperformed the sever which did not use monitoring information, by nearly a factor five in regards to workflow completion time. This is due, at least in part, to SPHINX’s ability to intelligently choose job placements which minimise the time spent waiting idle in the remote CE queue (as can be seen in the two right-hand histograms).

These results indicate that the current SPHINX prototype functions as expected, exploiting real-time knowledge of the grid, and demonstrates that SPHINX can successfully schedule work across a large number of grid sites that are owned by multiple units in a virtual organization.

CONCLUSIONS AND FUTURE WORK

A novel grid scheduling framework for computing has been presented in this paper. Resource scheduling is a critical issue in executing large-scale data intensive applications in a grid. This document outlines several important characteristics of a grid scheduling framework including dynamic workflow planning, enforcement of policy and Quality of Service requirements, and a flexible distributed, fault tolerant system.

SPHINX currently implements many of the characteristics outlined above and provides distinct functionalities, such as dynamic workflow planning and just-in-time scheduling in a grid environment. It leverages existing monitoring (MonALISA [7]) and execution management systems (VDT Client [3]). In addition, the highly customizable client-server framework can easily accommodate user specific functionality. This is due to a flexible architecture that allows for the concurrent development of modules that can effectively manipulate a common representation for the application workflows. The workflows are stored persistently in a database using this representation.

The development of SPHINX is still in progress and we plan a public beta release in the near term. As development progresses, we aim to include several additional core functionalities for grid scheduling. One such important functionality will be forecasting grid weather with estimated forecast uncertainties, enabling SPHINX to probabilistically assign jobs to resources within some future time window.

ACKNOWLEDGEMENTS

We would like to express our thanks to Dimitri Bourilkov and Craig Prescott for their assistance and insight. In addition, we would like to thank Caltech and in particular Conrad Steenberg, Mike Thomas, and Frank van Lingen for many fruitful discussions. Finally, we are indebted to Grid3 for the use of those resources for our experiments. This research was supported by the NSF under contract ITR 0086044 and ITR ASC-0325459

REFERENCES

[1] Foster, I., Kesselman, C., Tuecke, S. “The Anatomy of the Grid: Enabling Scalable VOs.” International J. Supercomputing Appliations, 15(3), 2001.

[2] http://clarens.sourceforge.net

[3] http://www.griphyn.org/vdt

[4] Foster, I., Voeckler, J., Wilde, M., Zhao, Y. “Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation.” 14th International Conference on Scientific and Statistical Database Management (SSDBM), 2002.

[5] In, J., Avery, P., Cavanaugh, R., Ranka, S. “Policy Based Scheduling for simple QoS in Grid Computing.” International Parallel & Distributed Symposium (IPDPS), Santa Fe, New Mexico, 2004

[6] http://www.ivdgl.org/grid3

[7] http://monalisa.cacr.caltech.edu