PREDICTING THE RESOURCE REQUIREMENTS OF A JOB SUBMISSION

Arshad Ali[4], Ashiq Anjum[4], Julian Bunn[1], Richard Cavanaugh[5], Frank van Lingen[1], Richard McClatchey[3], Muhammad Atif Mehmood[4], Harvey Newman[1], Conrad Steenberg[1], Michael Thomas[1], Ian Willers[2]

[1]*California Institute of Technology*
Pasadena, CA 91125, USA
Email: {fvlingen,newman,conrad,thomas}@hep.caltech.edu, Julian.Bunn@caltech.edu
[2]*CERN, Geneva, Switzerland*
Email: Ian.Willers@cern.ch
[3]*University of the West of England*
Bristol, UK
Email: Richard.mcclatchey@uwe.ac.uk
[4]*National University of Sciences and Technology*
Rawalpindi, Pakistan
Email: {arshad.ali, ashiq.anjum,atif.mehmood}@niit.edu.pk
[5]*University of Florida, USA*
Email: cavanaug@phys.ufl.edu

Abstract

Grid computing aims to provide an infrastructure for distributed problem solving in dynamic virtual organizations. It is gaining interest among many scientific disciplines as well as the industrial community. However, current grid solutions still require highly trained programmers with expertise in networking, high-performance computing, and operating systems. One of the big issues in full-scale usage of a grid is matching the resource requirements of job submission to the resources available on the grid. Resource brokers and job schedulers must make estimates of the resource usage of job submissions in order to ensure efficient use of grid resources. We propose a prediction engine that will operate as part of a grid scheduler. This prediction engine will provide estimates of the resources required by job submission based upon historical information. This paper presents the need for such a prediction engine and discusses two approaches for history based estimation.

INTRODUCTION

Grid schedulers are responsible for allocating distributed resources for job execution. To meet the user requirements the scheduler must select the resources that are capable of completing the job within constraints provided by the user. For the purposes of this paper we will consider the constraint of minimal job execution time. This imposes the requirement that the grid scheduler has the ability to estimate the execution time of a job on all available execution sites and then to select the one that has least estimated runtime. Our proposed module prediction engine will be a part of such a scheduler. In this paper we will discuss the need for estimation and offer a history based approach for estimating the run time of job submissions. Based on this approach two models for history maintenance will be discussed and a proposal for a runtime estimator will be presented.

NEED FOR PREDICTION

Job submissions on the grid are often represented by a set of interdependent tasks arranged in a Directed Acyclic Graph (DAG). The nodes of the DAG are the individual tasks and the branches are the dependencies between these tasks. The initial step for job submission involves the generation of this DAG. The individual tasks, along with their logical input and output filenames, and the dependencies between these tasks are identified. The result is an abstract workflow (AW) which identifies the resource needs of the job. This process of abstract workflow generation can be performed by a workflow planner such as the Chimera virtual data system [1]. After the creation of the DAG the resources identified in the AW must be mapped onto the available grid resources through a process of resource discovery and selection. This task is performed by a concrete job planner such as Pegasus [2]. The result is a concrete workflow (CW) which is submitted to a job executor such as DAGMan [3]. The transition from an abstract to a concrete workflow requires a prediction engine module that helps in selecting the optimum site or sites on which the tasks in the AW should be run. The prediction engine will assist the concrete job planner by selecting the optimum execution site. For example, the prediction engine might obtain load information at various available sites (through historical data or monitoring services like MonALISA[4]) to estimate the run time and resource consumption of the job. Based on these two pieces of information the

prediction engine will select the optimum site for executing the job, i.e. the one that has the least load after job scheduling and that will result in the shortest runtime.

The planner can utilize early or late binding of resources to the job plan. With early binding the planner makes an exact plan of computation based on the current information about the system. It decides where the tasks need to execute and the exact location from where the input data needs to be accessed. The prediction engine would assist the planner in making these decisions. Similarly the prediction engine will assist in selecting the replica location site with least transfer time. The planner would then map the abstract workflow onto the resources at the selected site. A concrete workflow would be generated and passed to the job executor (Condor-G/DAGMan).

With late binding the planner delays the process of mapping the task's resources to concrete grid resources until the task is ready to execute. At the time the executor is ready to run the task or perform data movements, the executor can consult the information services and make local planning decisions. Here also the prediction engine would assist the planner in selecting the site executor where the abstract work flow will be submitted.

The prediction will be part of the planning process by facilitating efficient planning in performing the following tasks:

- Predicting runtime time and resource (CPU, bandwidth, and memory) requirements of a job.
- Predicting data transfer time from the available file replica locations.

PREDICTING RUNTIME OF A JOB

Shonali, Seng and Arkady at Monsah University have proposed a history based approach for run time prediction [5]. This history based approach operates on the principle that applications with similar characteristics have similar runtimes. This requires maintaining a history of applications that have executed along with their respective runtimes. To estimate a future application's runtime, similar applications in the history are identified and a statistical estimate (such as the mean and linear regression) of their runtimes is computed.

The fundamental problem with history based runtime estimation is the definition of *similarity*; diverse views exist on the criteria that make two applications similar. For instance, we can say that two applications are similar because the same user on the same machine submitted them or because they have the same application name and are required to operate on the same-sized data. Thus, we must develop techniques that can effectively identify similar applications. Such techniques must be able to accurately choose applications' attributes that best determine similarity. The obvious test for such techniques is to measure the prediction accuracy of the estimates we obtain by computing a statistical estimate of the runtimes of the applications identified as similar. Thus, the closer the predicted runtime is to the actual runtime, the better is the technique's prediction accuracy.

The history based algorithm for run time estimation is based on Rough Sets [6] and operates as follows:

1. Maintain a history of the jobs that have executed along with their respective run times. Partition this history into decision and condition attributes [6]. Condition attributes will be various job parameters such as job owner, the queue to which the job was allocated etc and decision attribute will be the job runtime.
2. Compute equivalence classes [6] with respect to all condition attributes; these will be called condition relative equivalence classes.
3. Compute equivalence classes with respect to all decision attribute; these will be called decision relative equivalence classes.
4. Compute the positive boundary region [6] with respect to all condition attributes; this will be the union of all condition relative equivalence classes which are the subset of any decision relative equivalence class.
5. Compute the initial dependency of all condition attributes by using the positive boundary region computed in the previous step.
6. Compute the significance of each condition attribute.
    6-1. Compute the condition relative equivalence classes by removing the attribute (whose significance is to be computed) from the list of condition attributes.
    6-2. Compute the positive boundary region and dependency with respect to the resultant condition relative equivalence classes.
    6-3. Subtract the dependency computed in the last step from the initial dependency of condition attributes computed in step 5; this will be the significance of current condition attribute.
7. Compute the core [6] of the information system, this will be minimal set of condition attributes which are necessary for distinguishing objects in information system.
8. Compute reduct [6]. A reduct is a minimal set of attributes from Q (the whole attributes set) that preserves the partitioning of information system and therefore the original classes, in other words reduct consists of only the non redundant condition attributes.
    8-1. Subtract core from the set of all condition attributes.
    8-2. Sort the resultant set of condition attributes in descending order of significance.
    8-3. Add the condition attribute with highest significance from the sorted list computed in the last step in core. Repeat this process for the entire sorted list until the dependency of

core becomes equivalent to the initial dependency of condition attributes.

8-4. Remove redundant condition attributes from reduct.

8-4-1. Iteratively remove the non core attributes from reduct in ascending order of significance. If the dependency of reduct remains unchanged the attribute is considered redundant and is removed from reduct.

9. The reduct obtained from the last step is called the similarity template (ST) on the basis of which we will compare applications in history. The similarity template will consist of condition attributes that have a strong dependency with the decision attribute (runtime) i.e. the condition attributes that directly effect runtime.
10. Combine the current task T (i.e. for which the run time is to be estimated) with the history H to obtain new history H^T.
11. Apply the similarity template ST from step 2 on the history H^T. This will partition the history in to various equivalence classes.
12. Identify the equivalence class EQ^T to which the job T belongs.
13. Compute the statistical estimate (mean) of the run time of objects belonging to EQ^T, this will be the estimated runtime for input job T.

HISTORY MAINTENANCE

There are two ways to store the history information from job execution. It can be stored in a central job history database, or it can be decentralized with each execution site maintaining its own job execution history.

Centralized History for all execution sites

In the centralized history approach the prediction engine itself will maintain a central history of the jobs that are being scheduled and completed successfully. There will be a single history for the entire virtual organization (VO) partition controlled by a grid scheduler. When a job arrives for scheduling, the scheduler will pass it to the prediction engine module. The prediction engine will then analyze the centralized job history data to identify similar jobs. The estimated runtime of the requested job will be calculated based on the runtime of similar jobs.

Decentralized History at each Execution Site

In the decentralized history approach the gatekeeper node of every execution site will maintain the history of jobs that were scheduled and successfully executed on that site. The gatekeeper will contain a web service module that provides this information to remote grid schedulers. This predictive web service should provide a method that takes job condition attributes as input and returns the estimated run time as output. When a job arrives for scheduling the scheduler will pass it to the prediction engine, which will then contact the gatekeeper node at each execution site. The condition attributes comprising the similarity template will then be passed to the predictive web service which will estimate the run time of the job based on the history maintained at the site. This estimated time will then be returned back to the prediction engine. After receiving the estimated times from all available sites the prediction engine can then easily make a decision about the optimum site in terms of the least estimated runtime.

IMPLEMENTATION AND T ESTING

We are currently working on a prediction engine module. We have decided to use the decentralized history based approach because of two reasons:

1. For any job, the run time will be self estimated by each execution site taking into account the computational resources available at that site.
2. The percentage accuracy of the estimated time will be greater, since each execution site has its own history of previous runs rather than a single history for all execution sites.

The predictive service will be implemented as a web service hosted on a Clarens [7] web service host. The Clarens server runs on the gatekeeper node of the execution sites and also hosts the execution service. When a job arrives at the scheduler the following sequence of events will take place:

1. The planner will create an abstract workflow.
2. This workflow will be given to the prediction engine.
3. The prediction engine will contact available execution sites, and will pass the job attributes that constitute the similarity template to the Clarens server at every execution site.
4. The Clarens server will then estimate the run time of job.
5. The estimated run time of step 4 will then be returned back to the scheduler.
6. After getting estimates from every execution site, the prediction engine will then contact the MonALISA repository to get the status of the load at execution sites.
7. Based on the estimated run time and load status from step 6 and 7 the prediction engine will select a site that has the least predicted runtime i.e. product of site load and combined estimated run time queue wait time.
8. The prediction engine will tell the planner about the selected site. The planner will then map the abstract work flow on to the selected site.

Two modules are required for runtime prediction:

1. The prediction Engine, which will be part of the grid scheduler.
2. The Runtime Estimator, which will be a web service method inside of Clarens.

We have implemented the Runtime Estimator as part of the Clarens shell service. In order to test the Runtime Estimator we have used accounting data from the Paragon Supercomputer at the San Diego Supercomputing Center. This data was collected by Allen Downey in 1995. The accounting data had the following information recorded for each job: account name; login name; partition to which the job was allocated; the number of nodes for the job; the job type (batch or interactive); the job status (successful or not); the number of requested CPU hours; the name of the queue to which the job was allocated; the rate of charge for CPU hours and idle hours; and the task's duration in terms of when it was submitted, started, and completed.

The history consisted of 100 jobs and the runtime for 20 jobs was estimated; figure 1 shows the log of estimated and actual runtimes in each of the 20 cases:

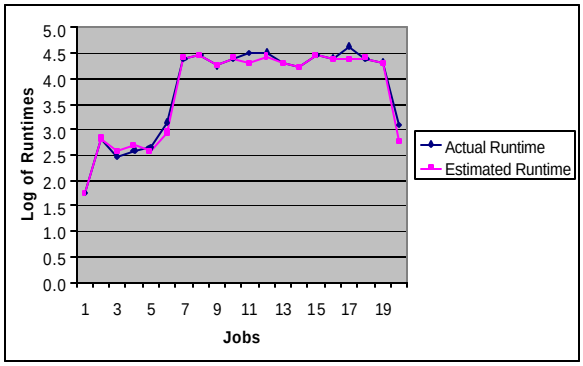

Figure 1: Actual & Estimated Runtimes for 20 test cases

For each of the twenty cases, we computed the error in estimation as:

$$\textit{Percentage Error} = ((\textit{ActualRuntime} - \textit{EstimatedRuntime}) \div \textit{ActualRuntime}) \times 100$$

The percentage errors for the twenty cases were then used to compute the mean error of the runtime estimator. The mean error for the run time estimator comes out to be 13.53%, this was computed by dividing the sum of percentage errors in each of the twenty test cases by 20.

CONCLUSION

Our results show that the history based approach for runtime estimation works with an accuracy of almost 80%, so the runtime estimator can assist the prediction engine in the selection of an optimum site. In future we plan to implement the Prediction Engine as part of the runtime estimation and to integrate it with the runtime estimator.

REFERENCES

[1] Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation Ian Foster1, Jens Vöckler, Michael Wilde, Yong Zhao.

[2] Pegasus: Mapping Scientific Workflows onto the Grid, Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, Miron Livny, Across Grids Conference 2004, Nicosia, Cyprus

[3] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Condor and the Grid", in Fran Berman, Anthony J.G. Hey, Geoffrey Fox, editors, Grid Computing: Making the Global Infrastructure a Reality, John Wiley, 2003. ISBN: 0-470-85319-0

[4] H.B. Newman, I.C. Legrand, P.Galvez, R. Voicu, C. Cirstoiu: "MonALISA: A Distributed Monitoring Service": CHEP 2003, La Jolla, California, March 2003

[5] Shonali Krishnaswamy, Seng Wai Loke, Arkady Zaslavsky "Estimating Computation Times of Data-Intensive Applications" IEEE DISTRIBUTED SYSTEMS ONLINE Vol. 5, No. 4; April 2004

[6] Rough Sets: A Tutorial Jan Komorowski, Lech Polkowski, Andrzej Skowron http://www.folli.uva.nl/CD/1999/library/pdf/skowron.pdf

[7] The Clarens Web Services Architecture Authors: (Conrad D. Steenberg and Eric Aslakson, Julian J. Bunn, Harvey B. Newman, Michael Thomas, Frank van Lingen) http://clarens.sourceforge.net/index.php/docs