A Case for Application-Aware Grid Services

Gabriele Garzoglio, Andrew Baranovski, Parag Mhashilkar, Anoop Rajendra*, Ljubomir Perković**
Computing Division, Fermilab, Batavia, IL; * University of Texas at Arlington, TX
** School of Computer Science Telecommunications and Information Systems, DePaul University, Chicago, IL

INTRODUCTION

In 2005, the DZero Data Reconstruction project processed 250 tera-bytes of data on the Grid, using 1,600 CPU-years of computing cycles in 6 months. The large computational task required a high-level of refinement of the SAM-Grid system, the integrated data, job, and information management infrastructure of the RunII experiments at Fermilab. The success of the project was in part due to the ability of the SAM-Grid to adapt to the local configuration of the resources and services at the participating sites. A key aspect of such adaptation was coordinating the resource usage in order to optimize the typical access patterns of the DZero reprocessing application. Examples of such optimizations include database access, data storage access, and worker nodes allocation and utilization.

A popular approach to implement resource coordination on the grid is developing services that understand application requirements and preferences in terms of abstract quantities e.g. required CPU cycles or data access pattern characteristics. On the other hand, as of today, it is still difficult to implement real-life resource optimizations using such level of abstraction. First, this approach assumes maximum knowledge of the resource/service interfaces from the users and the applications. Second, it requires a high level of maturity for the grid interfaces. To overcome these difficulties, the SAM-Grid provides resource optimization implementing application-aware grid services. For a known application, such services can act in concert maximizing the efficiency of the resource usage. We describe what optimizations the SAM-Grid framework had to provide to serve the DZero reconstruction and montecarlo production. We also show how application-aware grid services fulfill the task.

OPTIMIZATION PROBLEMS

High energy physics applications have different resource utilization requirements. The SAM-Grid meta-computing infrastructure is often used to run montecarlo and data reconstruction (data filtering) for the DZero experiment at Fermilab.

Activity	Description	Community	Load	time/job
Reconstruction	data filtering	Small	CPU & I/O	10 hours
Montecarlo	data simulation	Small	CPU	10 hours
Analysis	data mining	Large	CPU & I/O	hours to days

Activity	Input/Job	Output/Job	Input/Year	Output/Year
Reconstruction	GB	GB	100 TB	100 TB
Montecarlo	None	10 GB	Noue	TB
Analysis	100 GB	GB	varies	varies

Comparison of different characteristics among three typical computation activities of the DZero experiment. The bottom table focuses on the input/output data size. The numbers represent the order of magnitude.

Even restricting our system to manage resources for montecarlo generation and data reconstruction only, it was still a challenge to run efficiently jobs with such different characteristics. In order to let the grid organize the usage of the resources efficiently, we decided to expose details of the applications to the grid.

We present a few examples where the knowledge of the application helps the grid optimize the resource utilization. We use these examples to show that application-specific knowledge helps grid services optimize resources and run grid jobs efficiently.

(1) DATABASE ACCESS PROBLEM

Grid jobs submitted to an execution site are split into multiple parallel instances of the same application by the SAM-Grid grid-to-fabric interface. This typically results in dozens to hundreds of jobs starting approximately at the same time and, therefore, accessing key resources essentially concurrently.

In practice, not all the services have the same degree of accessibility. In particular for montecarlo generation, the parameters describing what type of physics to generate were accessed from a central database, which initially responded with a "denial of service" to 40% of the jobs. Introducing retrial with randomized exponential back off reduced the final job failure rate to 5%. Despite the reduced failure rate, grid jobs and their retrials increased the load of the database to a point where interactive access was extremely inconvenient (minutes per query).

This problem was properly solved by informing the grid of the database access characteristics of the montecarlo application. All the hundreds of jobs submitted by the grid, in fact, were parallel replicas of a single grid job and, therefore, required access to the same input parameters from the database.

The grid-to-fabric interface was enhanced to perform a single database access per grid job, when the job entered the site. The information was saved and redistributed to the parallel jobs by internal cluster transport mechanisms. This solution reduced the "denial of service" failure rate to essentially 0% and still maintained a high availability for interactive database accesses.

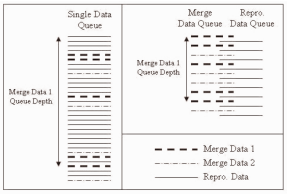
In conclusion, access to a grid resource (the database) was optimized by instructing grid components (the grid-to-fabric interface) of the characteristics of the application (parallel jobs requiring the same input parameters).

(2) DATA STORAGE ACCESS PROBLEM

Different applications have different typical input data access patterns. For DZero, data reconstruction applications begin data processing when a single input file, typically 1 Gigabyte in size, is delivered to the worker node. Instead, data merging applications, used in production operations to concatenate files typically 200 Megabytes in size, begin processing when multiple "small" input files are delivered to the worker node. Optimizing access to the storage resources with such different regimes is a concern.

In the SAM-Grid, applications transfer files from storage services that maintain queues of data access requests. The storage services, in fact, control their load by granting access to the data transfer servers a few requests at the time. Access to a transfer server is granted in the order in which the access request is submitted. When reconstruction and merging applications use the same data queue to access their input, transfer requests for the various input files are interleaved. This leads to inefficiencies, because in real life, on a cluster, reconstruction jobs are one or two order of magnitude more abundant than merging jobs. This means that requests for each input file of a merging application is interleaved with a dozen input files of reconstruction applications. Therefore, before starting processing data, a merging application often needs to wait for these multiple reconstruction transfers to occur, thus substantially increasing its idle time. For a cluster of 900 CPU, this idle time is between one and two hours.

This inefficiency can be reduced by instructing the grid of the input data access characteristics of the application. Knowing the number of required input data files, the data storage service can organize requests from reconstruction applications in a queue different from the requests from merging applications. Thus, a few merging applications only compete amongst themselves for file access, drastically reducing their idle time.



A diagram representing queues of requests for file access. On the left, a single queue manages requests from reprocessing jobs (straight lines), and merging jobs (dashed and dashed-dotted lines). Reprocessing jobs are two orders of magnitude more abundant than merging jobs. Merging job 1 needs to access five input files before it can start running (dashed lines, bold for clarity). On the right, requests from merging and reprocessing jobs are managed by two different queues. If access requests are granted one at the time, the queue depth for merging job 1 is much shorter than in the case of the single queue (left diagram). If the data storage server knows the typical data access pattern of the jobs, it can optimize access to the data. The SAM-Grid storage elements have knowledge of the typical data access patterns of each application.

In conclusion, as in the previous, example access to grid resources (data files) was optimized by instructing grid components (the storage service) of the characteristics of the application (multiple or single input data requirement).

(3) WORKER NODES ALLOCATION PROBLEM

The grid-to-fabric interface of the SAMGrid submits multiple batch jobs for every grid job entering the site. How many worker nodes should be allocated for a given application? In general, to accomplish the same amount of computation for a grid job, the fewer batch jobs are submitted, the longer each job runs, and vice versa. There is an acceptable range for the running time of a job. Batch jobs should not run too long to minimize the probability of termination before completion. Jobs are typically terminated because they run beyond the maximum wall-clock time allowed by the local scheduler, or because they are evicted due to a higher-priority job entering the scheduler, or because of hardware failures. On the other hand, batch jobs should not run too short in order to maximize the ratio between running time and setup time i.e. the time needed to prepare the job environment (in the SAM-Grid typically around half an hour).

The "suitable" expected running time is managed by the grid controlling the number of worker nodes allocated for running the application. It should be noted that applications may have additional constraints on the number of jobs. These constraints are dictated by considerations on ease of bookkeeping and of recovery after failures. In any case, the number of worker nodes to allocate depends on the type of application. For reconstruction applications, the grid-to-fabric interface allocates a worker node for every file in the dataset specified for the grid job. Given the computational requirements of the reconstruction application, this approach gives an acceptable running time of a few hours on a modern CPU and eases bookkeeping and recovery operations. For montecarlo applications, the interface computes the number of worker nodes to be allocated by dividing the total number of events to be produced as specified for the grid job by the "optimal" number of events per job. The "optimal" number of events is a parameter configured at the site, considering the speed of the average CPU at the site, the computational requirements of the montecarlo application, and other scheduler constraints (maximum allowed wallclock time, etc.).

In conclusion, as in the previous examples, allocation of grid resources (worker nodes) is optimized by instructing grid components (the grid-to-fabric interface) of the characteristics of the application (computational requirements of the application and other constraints).

(4) MINIMAL RESOURCE IDLE TIME PROBLEM

Grid jobs are often internally composed of interdependent tasks. We let the grid manage the order of execution of each internal task/job automatically. This automation minimizes the idle time between job submissions, thus minimizing the idle time of the resources.

In order to decide whether to submit a job, the grid must be able to determine whether the jobs on which it depends were successfully executed. In general, determining the success of a job is a difficult task. In case of complex computational activities, success is generally never defined only by the exit status of the job. To determine whether a montecarlo generation job was successful, for example, the grid has to check the number of events produced by the job by querying a bookkeeping database and compare this number with the number of events originally requested. Success is determined by policy: typically, if more than 90% of the events have been produced, the job is successful. For reconstruction applications the success policy is defined differently: typically a job is successful only if it has reconstructed all the input files, unless subsequent recovery jobs fail twice on the same event with the same error, thus exposing a corrupted input file. At any rate, having the grid determine the success of a job is an application-specific task.

In conclusion, as in the previous examples, the idle time of grid resources is minimized by instructing grid components (the job management component) of the characteristics of the application (policy defining the success condition).

CONCLUSIONS

Application-specific knowledge is important in the optimization of grid resources. Two approaches are possible:
1) Applications communicate their requirements and preferences in terms of abstract resource/service-specific quantities. This is difficult to achieve as it requires a very high level of maturity of the grid interfaces and a thorough understanding of application requirements.
2) Applications rely on Application-Aware Grid Services for resource optimizations. This is less general but easier to implement and extend.

The SAM-Grid used successfully Application-Aware Grid Services for grid resource optimization.