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 pre-supposes the knowledge of the resource/service interfaces from the user application, which 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 set the access node, reducing the efficiency of the resource usage. We describe what optimizations the SAM-Grid framework had to provide to solve the DZero reconstruction and monte-carlo production. We also show how application-aware grid services fulfill the task.

Even restricting our system to manage resources for monte-carlo 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 oriented to the SAM-Grid grid-to-interface. This typically results in dozens to hundreds of jobs starting approximately at the same time and, therefore, accessing key resources: data, storage, and CPU. 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.

In practice, not all the services have the same degree of accessibility. In particular for monte-carlo generation, the percentage of data in the "type of physics" to which were accessed from a central database, which initially responded with a "database of service" to 40% of the jobs. Introducing retrieval with randomization exponential backoff reduced the final job failure rate to 5%. Despite the reduced failure rate, grid jobs and their retrieval increased the load of the database to a point where interactive access was extremely inconvenient (minutes per query).

In conclusion, as in the previous example, access to grid resources (data files) 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 and reconstruction applications begin data processing when a single input file, typically 1/10th of a GB, is delivered to the worker node. Instead, data merging applications, used in production operations to concateenate files typically 200 Megabytes in size, when multiple "small" input files are delivered to the worker node. Optimizing access to these 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 queues 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 those 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 server can request a few requests from reconstruction applications in a queue different from the requests from merging applications. As a result, a few merging applications complete amongst themselves for file access, drastically reducing their idle time.

This problem was properly solved by informing the grid of the database access characteristics of the monte-carlo application.

The grid-to-fabric interface of the SAM-Grid submits multiple batch jobs for every grid job entering the site. Therefore, many worker nodes should be allocated for each 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. Therefore, there is an acceptable rate for the running time of a job. Batch jobs should not run too long the storage server, because they are evicted due to a higher-priority job entering the scheduler, or because of hardware limitations. Instead, batch jobs should not run too short in order to minimize the ratio between running time and setup time i.e. the time needed to prepare the environment (in the SAM-Grid typically around half an hour).

The "expected" 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 booking and of recovering after failures. In any case, the number of worker nodes to allocate is determined by the characteristics of the 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 unacceptable number of running workers on a modern CPU and eases bookkeeping and recovery operations. For monte-carlo applications, the interface establishes the number of worker nodes to allocate by dividing the total number of events to be produced as specified for the grid job by the "specific 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 monte-carlo application, and other scheduler constraints (maximum allowed wallclock time, etc.).

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

(3) WORKER NODES ALLOCATION 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 maximizing 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 because of the complexity of computational activities, success is generally never defined only by the exit status of the job. To determine whether a monte-carlo 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 important service.

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.