Customized Plug-in Modules in Metascheduler Community Scheduler Framework 4 (CSF4) for Life Sciences Applications

Zhaohui DING[1], Xiaohui WEI[1], Yuan LUO[1], Da MA[1], Peter W. ARZBERGER[2,4], Wilfred W. LI[3,4]

College of Computer Science & Technology, Jilin University, Changchun, Jilin, China 130012[1], Center for Research in Biological Systems[2], San Diego Supercomputer Center[3], University of California, San Diego[4], La Jolla, CA 92093

Abstract: As more and more life science researchers start to take advantages of grid technologies in their work, the demand increases for a robust yet easy to use metascheduler or resource broker. In this paper, we have extended the metascheduler CSF4 by providing a Virtual Job Model (VJM) to synchronize the resource co-allocation for cross-domain parallel jobs. The VJM eliminates dead-locks and improves resource usage for multi-cluster parallel applications compiled with MPICH-G2. Taking advantage of the extensible scheduler plug-in model of CSF4, one may develop customized metascheduling policies for life sciences applications. As an example, an array-job scheduler plug-in is developed for pleasantly parallel applications such as AutoDock and Blast. The performance of the VJM is evaluated through experiments with mpiBLAST-g2 using a Gfarm data grid testbed. Furthermore, a CSF4 portlet has been released to provide a graphical user interface for transparent grid access, with the use of Gfarm for data staging and simplified data management. The platform is open source at sourceforge.net/projects/gcsf/ and has been deployed in life science gateways by projects such as My WorkSphere, and PRAGMA Biosciences Portal. The VJM enables the development of support for more sophisticated workflows and metascheduling policies in the near future.

Key words: Metascheduling, MPICH-G2, co-allocation, scheduler plug-in, life sciences

1 Introduction

1.1 Background

Applications in systematic modeling of biological processes across scales of time and length demand more and more sophisticated algorithms and larger and longer simulations. The emerging grid computing technologies are enabling the creation of virtual organizations and enterprises for sharing distributed resources to solve large-scale problems in many research fields. More and more bio-scientists start to take advantage of grid technologies to facilitate their research. Metascheduling is able to provide a virtualized interface for the end users to access heterogeneous grid services and resources transparently (see section 1.3). However, there are still many stumbling blocks in metascheduling to attract more highly focused biomedical researchers to use grid computing to solve their concrete problems.

First of all, running parallel jobs crossing domains in a grid environment is still a challenge. Parallel Virtual Machine (PVM) [1] and Message Passing Interface (MPI) [2] are usually used in the traditional single cluster environment. Grid enabled MPICH (MPICH-G2) [3] is a Grid-enabled implementation of MPI, which is able to execute a parallel job across multiple domains and has been used in many life science applications. Nonetheless, MPICH-G2 does not synchronize the resource allocation in multiple clusters, which would cause resource waste or even dead lock problems (see section 3.1). Many researchers have found that the MPICH-G2 jobs cannot execute successfully unless manual reservations are made in advance

at destination clusters. Apparently, without an automated high-level coordinator's participation, these problems are hard to resolve. In this paper, we introduce a virtual job model (VJM) for metascheduling, which is able to synchronize the cross-domain resource allocation for parallel jobs, and eliminate the resource dead lock mentioned above. Moreover, VJM does not rely on resource reservation and can work with any local scheduler which supports GRAM. Backfilling is also provided in VJM to alleviate resource waste. VJM is easy to deploy requiring no changes to the MPICH-G2 library or local schedulers.

It is well known that intelligent and suitable policies can improve the performance of a distributed system remarkably. However, the need to develop generalized and reusable scheduling components for cyberinfrastructure to the general scientific community and the desire for customized solutions by biomedical application scientists create a dilemma. Such a challenge exists in a single cluster as well, but it is much harder to resolve in the grid environment, because the users from different domains have diverse requirements, let alone the difficulty to evaluate the impact of local clusters' work load and policies on metascheduling. In this paper, we develop additional plug-in modules for the Community Scheduler Framework (CSF4) [4], in particular, an array-job plug-in to schedule AutoDock [5] or Blast [6] like applications, and demonstrate that many popular life science applications can take the advantage of CSF4 metascheduling plug-in model.

Thirdly, biomedical domain scientists usually have little time to learn new programming models or computing technology, hence, minimizing the cost of entry to the grid is another primary issue to address. It has previously been shown that Gfarm-FUSE (File System in User Space) may enable legacy applications to leverage grid resources without modification [7]. As a further effort to ease the scheduling of application tasks to the different resources, we have developed a CSF4 Portlet as a graphical user interface for submitting and monitoring jobs on distributed resources. We believe such efforts are able to flatten the learning curve for computational bio-scientists and engage them to use the grid willingly.

1.2 Metaschedulers and Resource Brokers

Many kinds of grid resource brokers, metaschedulers and utilities have been proposed and developed for parallel computing and scheduling in the heterogeneous grid environment. They are quite different in resource management protocols, communication mode and applicable scenario, etc.

The Moab grid scheduler (Silver) consists of an optimized and advanced reservation based grid scheduler and scheduling policies, which guarantee the synchronous startup of parallel jobs by using advanced reservation [8]. Silver also proposed the concept of synchronized resource reservation for cross-domain parallel jobs. GARA [9] splits the process of resource co-allocation into two phases: reservation and allocation. The reservation created in the first phase guarantees that the subsequent allocation request will succeed. However, GARA did not address synchronized resource allocation. Furthermore, GARA and Silver both require that local schedulers support resource reservation. Many other metaschedulers, such as Gridway [10], have yet to make optimizations for cross-domain parallel jobs.

Many metaschedulers are built on the top of specific local resource management system, such as Silver and Condor-G [11]. They provide plenty of policies derived from local schedulers. However, these policies are implemented under specified local system protocols, so they are not available in a heterogeneous environment. The Nimrod/G resource broker introduces the concept of computational economy to metascheduling [12]. It leverages the Globus Monitoring and Discovering System (MDS) to aggregate resource information and enforces scheduling policies based on an auctioning mechanism. To process resource negotiation, it is required that resource brokers use a common protocol like SNAP [13], and the negotiation result is difficult to predict. Gridway's scheduling system follows the "greedy approach", implemented by the round-robin algorithm.

1.3 Metascheduling of Grid Resources

Even with a metascheduler, the jobs still has to go through local schedulers. The metascheduler selects the best clusters for job execution, and the local scheduler maps the jobs to specific hosts. Cluster selection matches the available resources with specific job requirements. However, metascheduling polices are not higher level duplications of local scheduling policies because the metascheduler is not the real resource owner. Hence, metascheduling needs to consider not only resource availability but also local scheduling policies of each cluster in the job scheduling algorithm.

On each host, there are resources available for job scheduling, such as OS, CPU, memory, etc. The host based resource set (HRS), is defined as all the available resources for a host.

$$\boldsymbol{HRS} = \{HR_1, HR_2, \ldots, HR_m, cusHR_1, \ldots, cusHR_n\} \quad (1)$$

Where, HR_i=1..m are the common resources on each hosts, like CPU, memory, disk etc. $cusHR_i$=1..n are the customized resources on each host. For example, if the license for commercial software is installed on a host, then the host gets a customized resource to execute the specific software. Normally, customized resources may not be understood by generic scheduling policies.

The availability of each resource can be evaluated by a value, which is normally called resource load. Resource load can be described as the pair of *resource* and *value*. Since a resource can be static, like host type; or dynamic, like available memory, whose values could vary. The *value* could be of different data types, float, integer, boolean or string (host type, for example), etc. The resource availability of a host can be evaluated by its host resource load set (HRLS),

$$\boldsymbol{HRLS} = \{(HR_1, Val_1), \ldots, (HR_m, Val_m), (cusHR_1, cusVal_1), \ldots, (cusHR_n, cusVal_n)\} \quad (2)$$

Accordingly, the cluster based available resources set (CRS) is defined as,

$$\boldsymbol{CRS} = \{HRS_1, \ldots HRS_n\} \quad (3)$$

Where n is the total host number of the cluster, and HRS_i is the HRS for host i. And the resource availability of a cluster can be evaluated by cluster based resource load set (CRLS),

$$\boldsymbol{CRLS} = \{HRLS_1, \ldots HRLSn\} \quad (4)$$

Where n is the total host number of the cluster, and HRLSi is the HRLS for host i.

The resource requirements of a job are normally condition expressions regarding resource loads, such as, mem > 256 MB (memory) && OS=Linux (operating system). At same time, the user will also give out other requirements regarding scheduling, like QoS (Quality of Service) criteria, priority, wall time, or job dependency and so on. Hence, a job's specification (JS) regarding to scheduling can be divided into the requirements related to physical resources and those related to information about the resources or the jobs as below,

$$\boldsymbol{JS} = \{Rsrc\text{-}Reqs, Non\text{-}Rsrc\text{-}Reqs\} \quad (5)$$

The metascheduling algorithm is to decide when, where and which clusters the jobs go based on JS (eq 4) and CRLS (eq 5). Therefore, the major functionality of a metascheduling algorithm can be summarized as the following three functions (**Fun**$_x$, where x is the abbreviated name attributed to the function):

Job sorting function

Fun$_{\textbf{job-sort}}$(job list, JS.Non-Rsrc-Reqs, meta-policy) → sorted job list

$Fun_{job\text{-}sort}$ decides the job execution order based on metascheduling policy and the job non-resource requirements, such as priority, wall time, etc.

Cluster matching function

Fun$_{\textbf{CL-match}}$(JS.Rsrc-Reqs, CRLSs) → matched cluster list

Based on CRLS of each cluster, $Fun_{rsc\text{-}match}$ selects clusters that satisfy the job's resource requirements.

Cluster sorting function

Fun$_{\textbf{CL-sort}}$(CRLSs, local policies, local work load, JS) → sorted matched cluster list

$Fun_{CL\text{-}sort}$ evaluates the matched clusters CRLS and policies, and decides which cluster is the best for the job, and which one is the second best and so on.

There is no big difference between metascheduling and local scheduling regarding $\mathbf{Fun_{job\text{-}sort}}$ and $\mathbf{Fun_{CL\text{-}match}}$. Some tools are already available to get a cluster's CRLS and the work load information in a cluster, such as Ganglia [14] or SCMSWeb [15]. Many features regarding to $\mathbf{Fun_{job\text{-}sort}}$ and $\mathbf{Fun_{CL\text{-}match}}$ can be copied from local schedulers with little change. However, $\mathbf{Fun_{CL\text{-}sort}}$ is a big challenge for metascheduling in most cases as the evaluation of a cluster is quite different from the evaluation of a host. First, it is not straightforward to evaluate a cluster's resource availability based on CRLS. For example, the average CPU load may not make much sense to compare two clusters as the clusters may have different host numbers, machine types. Secondly, it is very hard for metascheduler to evaluate how a cluster's local policy impacts the metascheduled jobs, e.g., default queues vs. special queues that may be dependent on the wall clock time of a job. A job forwarded to a lightly loaded cluster may wait for a longer time than the jobs forwarded to other clusters, if the lightly loaded cluster's local policy assigns a very low priority to external jobs. Furthermore, the local jobs of a cluster compete for resources with the metascheduler. Therefore, a metascheduler must consider the dynamic work load in local clusters and the possible conflicts with local scheduling decisions.

2 CSF4 Metascheduling Plug-in Model

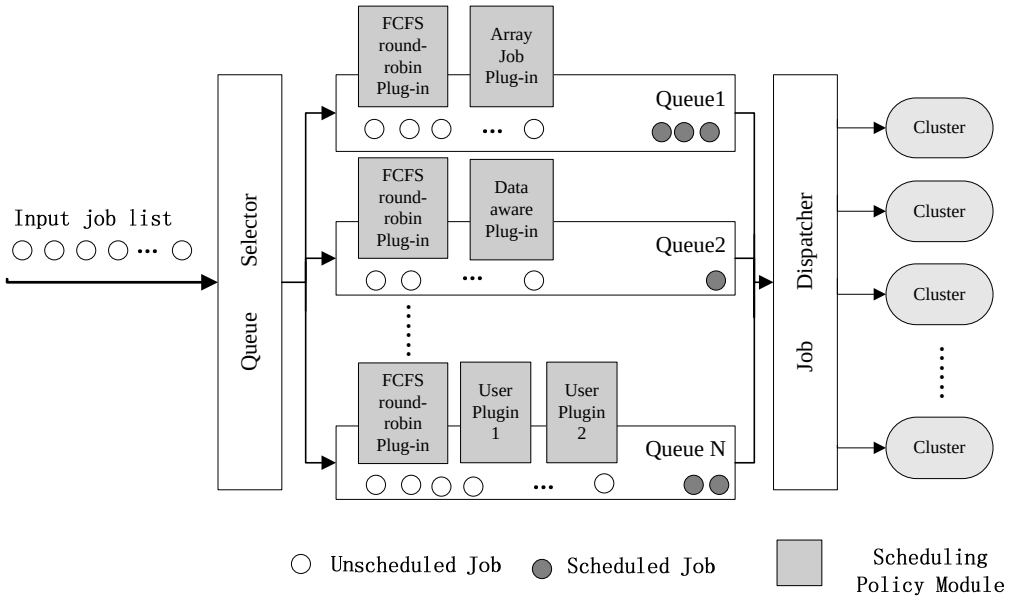


Figure 1 CSF4 Metascheduling plug-in Architecture

Although it is very hard to design generalized metascheduling strategies for the reasons above, system administrators maintaining metascheduler services could work out optimized metascheduling policies which fit the needs of their supported users. Moreover, customized resources may be added for consideration not supported by the scheduling policies out of the box. CSF4 has been designed to support a scheduler plug-in model to facilitate the implementation of customized scheduling policies.

Figure 1 depicts the metascheduler plug-in architecture implemented in CSF4. The key design principle is to isolate the functions of job management and resource load update, etc., from the scheduling policies, by encapsulating the specific scheduling policies as plug-in modules, and allow maximum flexibility in the configuration of scheduling policies when designing specific queue types. The plug-in model consists of the

plug-in framework and any number of plug-in modules. The plug-in framework maintains the metadata on jobs and resources and performs the functions of job submission, job dispatch, job lifecycle management, and resource update etc. Each plug-in module only consists of the logic to decide where and when to execute a job according to a specific policy.

2.1 Plug-in Module Development

Each plug-in module only consists of the following call back functions described in section 1.3 to be implemented by the developers/administrators. The framework is in charge of the control flow and data flow, and triggers these functions at proper times. The Initialize and Dispatch functions are provided by the framework.

Initialize(confFile[in]), which performs the initialization for the plug-in module, such as reading the configuration file (confFile) and initializing the internal data variables.

Fun$_{\textbf{job-sort}}$(job list[in/out], JS list[in]), which decides the job execution order based on metascheduling policy to be implemented and the non-resource requirements of a job. The function returns a sorted job list instead of a single job with the highest priority so that the other plug-in modules have the chance to adjust the job execute order.

Fun$_{\textbf{CL-match}}$(JS.Rsrc-Reqs[in], CRLSs[in], matched cluster list[out]), which selects the qualified clusters that satisfy the given job's resource requirements.

Fun$_{\textbf{CL-sort}}$({CRLSs, work load}of matched clusters [in], JS[in], execution cluster list[out]), which sorts the matched clusters and select the best clusters for the job execution. The function returns one or multiple clusters based on whether the jobs require cross cluster computations.

Dispatch (job[in], execution cluster list[in], dispatch decision[out]), which makes the final scheduling decision for the job.

The metascheduling policy to be implemented does not need to be an input parameter in any of the above functions. The administrator can configure multiple job queues with different policies via a queue configuration file (confFile). Each plug-in module is invoked in the order specified, when an end user submits its jobs to a desired queue. The job submission requires a Globus Resource Specification Language (RSL) file, using a simple command line invocation, such as 'csf-job-submit'. Currently, the local cluster policies do not appear in the functions. Instead, we provide a simple application programming interface (API) to access MDS for the above call back functions. The data in MDS typically include those provided by cluster monitoring software such as Ganglia and any local cluster policies supplied to MDS.

2.2 Array Job Plug-in For AutoDock and Blast-like applications

Quite frequently, life sciences applications are "pleasantly parallel", i.e., serial applications which may be used to handle many parallel data input. For example, AutoDock may be used to dock different ligands to a target protein structure, or Blast may be used with different input sequences to search for potentially related sequences within a target database. Here we show how a customized scheduling policy for these applications may be developed using the CSF4 plug-in model. A testbed of three clusters is setup with the Gfarm [16] deployed, each with different host numbers and dynamic local work load. Normally AutoDock or Blast applications consist of a large number of subjobs. These subjobs execute same binary with different input/output files, so the plug-in is named array job plug-in, similar to what's available for some local schedulers. The metascheduling objective is to balance the load between clusters and complete the entire set of jobs as soon as possible. The array job plug-in call back functions implemented are as follows:

Initialize() sets up the maximum job load, 10, for example, for all local clusters. If the number of unfinished subjobs in a cluster exceeds this maximum job load, the metascheduler will not send any new job

to it.

As the subjobs of AutoDock or Blast do not communicate to each other, and there is no dependency among them, the job execution order does not matter. Hence, **Fun**$_{\textbf{job-sort}}$() is just a empty function in the plug-in.

As the input/output files are accessible in all the clusters through the Gfarm virtual file system, so any cluster can run AutoDock or Blast jobs as long as its local scheduler is up. Therefore, **Fun**$_{\textbf{CL-match}}$() will simply select all clusters with local schedulers as matched clusters.

Fun$_{\textbf{CL-sort}}$() sorts the matched clusters according to their unfinished sub-job numbers. For example, in the beginning, the metascheduler dispatches 10 sub-jobs to both Cluster A and B. While doing the next scheduling, Cluster A finished 6 jobs and cluster B finished 4 jobs, then Cluster A would be preferred as the execution cluster for next job. If all the clusters have hit the maximum job load number set by **Initialize**(), no cluster is qualified as an execution cluster.

Dispatch() dispatches the first job in the job list to the execution cluster selected by **Fun**$_{\textbf{CL-sort}}$( ), and increase the cluster's unfinished job number.

Then, the metascheduler calls **Fun**$_{\textbf{job-sort}}$(), **Fun**$_{\textbf{CL-match}}$(), **Fun**$_{\textbf{CL-sort}}$() and **Dispatch**() again for the next job until no execution cluster is returned by **Fun**$_{\textbf{CL-sort}}$(). In the case above, cluster A gets 6 new jobs, and cluster B gets 4. Although the local cluster's load and policies are not considered explicitly, the loads are balanced dynamically since the cluster which completes jobs faster will get more jobs.

3 Virtual Job Model

To support cross-domain parallel jobs in different grids, we need to resolve three issues. First, the input or output files should be accessible in all the clusters where the job executes. Second, the inter-process communications of an application should be guaranteed between hosts in different domains. Last, the resource allocations in these clusters should be synchronized. The data grid technologies, like Gfarm, can be used to provide the global data availability. MPICH-G2 is a grid enabled MPI implementation to enable a user to run MPI programs across multiple sites. However, as MPICH-G2 does not have the functionality to synchronize the resource allocation at different sites, the parallel jobs may fail to start due to dead locks. Manual reservation may be used at different sites but is not scalable. In this section, we discuss the Virtual Job Model (VJM) to realize synchronized resource co-allocation for parallel jobs.

3.1 Existing problems of MPICH-G2

When user submits a MPI parallel job by *mpirun*, *mpirun* parses the job request first. Based on pre-defined information in a configuration file or MDS, *mpirun* decides the job execution clusters using round-robin policy and generates a RSL (Resource Specification Language) script. Then *globusrun* is used to distribute the subjobs to the clusters specified in the RSL file. As such, MPICH-G2 is highly dependent on a separate batch scheduler and manual intervention to succeed. First, since there is no cluster availability check, MPICH-G2 could submit subjobs to a cluster which is not even online. Second, when the subjobs of a parallel job are submitted to local clusters, they are scheduled by local schedulers based on their local policies to compete for resources with local jobs instead of running immediately. Since MPICH-G2 does not have a built-in mechanism to support synchronized resource allocation, the jobs starting earlier have to wait for the other jobs still waiting for resource allocation, resulting in resource wasting. Lastly, when multiple parallel jobs were submitted, the resource allocation dead lock could occur due to the resource competition and various local scheduling policies. As shown in Figure 2, there are two clusters in MPI resource configuration file, *C1* and *C2*, each with only 1 node available. Job *a* and job *b* are both MPI jobs that require two nodes. If *mpirun* assigns *C1* and *C2* to the two jobs, and job *a*'s subjobs is represented as a_1 and a_2, and job b's subjobs as b_1 and b_2. The subjobs a_1 and a_2 are submitted to cluster C1 and C2 earlier than subjobs b_1 and b_2. However, due to different local policies, the local scheduler on *C1* may put a_1 ahead of b_1

while scheduler on *C2* may put a_2 behind b_2. Then a dead lock occurs until either job *a* or job *b* gets killed.

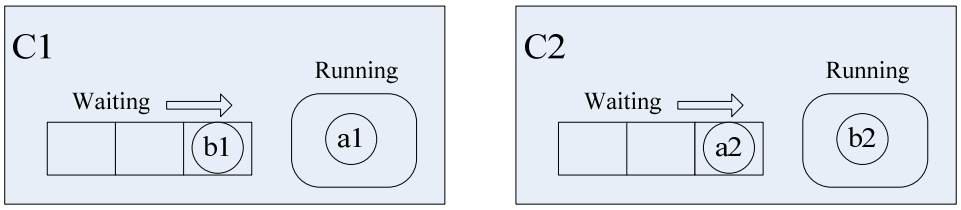


Figure 2 An example of resource dead lock of multiple parallel jobs

3.2 Design of Virtual Job Model (VJM)

VJM is a new metascheduling model designed to resolve existing problems for cross-domain parallel jobs in MPICH-G2. Before starting the actual job, VJM dispatches a virtual job to the different clusters to synchronize resource allocation. In consideration of the heterogeneity and autonomy of the grid environment, the key design principles of VJM are to require only minimal common features supported by all local resource managers, (no resource reservation), and to require no extra components or changes at any local site.

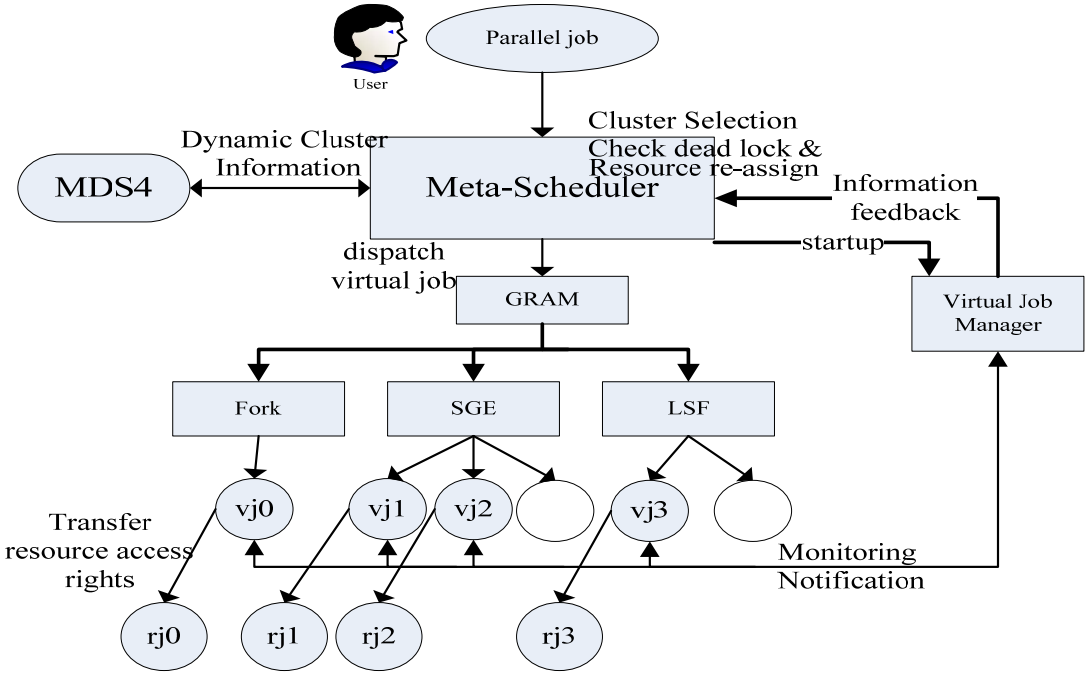


Figure 3 Virtual Job Model. Abbreviations: vj0, virtual job 0; rj0, real job 0.

With VJM, the users submit their MPICH-G2 jobs to the metascheduler instead of using *mpirun*. The metascheduler will be in charge of the resource allocation and job execution. VJM consists of three stages, resource availability check stage, resource allocation stage, and job startup stage (Figure 3).

3.3 Scheduling Stages of VJM

3.3.1 Algorithm Overview

First the metascheduler performs resource availability check before sending resource allocation requests to local schedulers. The pre-check first checks if a cluster is online, then it checks the resource requirements of the job and the resource availability of the cluster. Based on the pre-check results, the metascheduler decides

which clusters are qualified as execution clusters. With the pre-check, the jobs are never sent to the offline clusters, and a job with resource requirements that exceed the total resource capability of all clusters is rejected with a warning.

After the pre-check, the metascheduler makes a temporary decision about on which clusters to execute the parallel job and how to distribute the subjobs among them according its metascheduling policies. Since the metascheduler is not the owner of local cluster resources, it cannot allocate resources directly. Hence, a virtual job mechanism is introduced to co-allocate the resource for a parallel job.

At resource allocation stage, a virtual job is dispatched to the clusters instead of the real job. The virtual job consists of the same number of subjobs as the real parallel job and its responsibility is to obtain the guaranteed resources for the real job and update the resource allocation status in each cluster to the metascheduler. The metascheduler creates a virtual job manager (VJmgr) for each virtual job to collect the resource allocation information. Once a virtual job's subjob gets the resource and starts up, it will send a "READY" notification to VJmgr and wait for instructions from the VJmgr.

During the startup stage, after the VJmgr has received "READY" notification from all the virtual jobs, a "startup" instruction is sent to every virtual subjob to startup the corresponding real job's subjob. Since the virtual job can guarantee the resource availability, virtual job model can make sure that all the subjobs of a parallel job are synchronized.

Moreover, during the resource allocation stage, VJmgr is able to detect the potential dead lock and the cluster runtime error so that the metascheduler can adjust its scheduling decision accordingly. At the same time, VJmgr is able to backfill smaller jobs to the partially allocated resource of a larger parallel job to improve the resource usage of the system.

3.3.2 Deadlock Detection

All the parallel jobs are sorted on each cluster based on the resource allocation status by the metascheduler using the following rules,

For any two virtual jobs, VJ_a and VJ_b, if and only if VJ_a's subjobs on cluster C have been allocated the resources, and VJ_b's subjobs on cluster C cannot get the resources unless the resources allocated to VJ_a are released, we say $VJ_a < VJ_b$ on cluster C or $VJ_b > VJ_a$ on cluster C. Then, VJM uses the following algorithm to detect the resource allocation deadlock among virtual jobs (J),

$\exists (J_a, J_b, C_m, C_n) \rightarrow (J_a < J_b \text{ on } C_m, J_a > J_b \text{ on } C_n)$

If there exist a pair of clusters, C_m and C_n, the job $J_a < J_b$ on C_n and the job JB < JA on C_m, then VJM considers that there is a deadlock between J_a and J_b's resource allocation. Whenever VJM detects a deadlock, it will re-allocate either the resource for the subjobs of J_a on C_n or the resources for the subjobs of J_b on C_m to break the deadlock. The users can design different policies here, for example, lower priority, shorter wall clock time, or larger number of cpu's may be preferred.

Besides the resource dead lock, the run-time cluster unavailability (such as server down or very heavy workload) can also be detected by VJmgr. VJmgr regards a cluster as invalid for a virtual job when its subjobs cannot startup within a predefined time. Then the metascheduler will re-dispatch the subjobs to another cluster.

3.3.3 Backfilling

In a distributed system, backfilling is widely used while scheduling parallel jobs to optimize resource usage. VJmgr is also able to backfill smaller jobs to the partially allocated resource of a larger parallel job to alleviate resource wasting. Such smaller jobs may be effectively jobs that may finish within a fixed amount of wall clock time, and their resource requirements are met with the resources currently allocated. In a single cluster, the local scheduler has full control of its jobs and resources, so it can backfill any job to the

resources allocated for another job. In the grid environment, however, access control poses problems to the backfilling process. In VJM, the jobs are started by virtual subjobs, instead of the scheduler, which only have the access permission of the desired job owner. Therefore, only the jobs from the same user can be backfilled safely. Otherwise, the backfilled job may fail to access the system resources at run time, such as input/output files.

On the other hand, data grid products like Gfarm which uses user proxies instead of user id or user/password to authenticate an access request. This provides more flexibility to backfilling in VJM. In the implementation of VJM in CSF4, the metascheduler is able to delegate the user proxy of the backfilled job to the virtual jobs. Then, the virtual jobs can setup the correct credential context for those jobs to make sure they can access the Gfarm file systems successfully at runtime.

In summary, the VJM resolves the problems in resource co-allocation for MPICH-G2, takes advantages of the scheduling policy plugins in the metascheduler to optimize the cross-domain parallel job scheduling, and ensures proper backfilling of jobs when used with Gfarm.

4 Experiments

4.1 Evaluation of Virtual Job Model

VJM invokes additional processes for synchronized resource allocation, including startup of virtual jobs and virtual job managers, with communications between the two, all resulting in additional overheads. However, the pre-check mechanism, runtime resource availability, and resource dead lock resolution, and small job backfilling, all lead to improved parallel job throughput and resource usage.

4.1.1 Pre-check of resource availability

Resource availability pre-check is important in a heterogeneous grid environment. With a Gfarm testbed consisted of 4 different clusters, jobs requiring 3 out of 4 clusters (*cpi*, a MPI program for the calculation of *pi* , compiled with mpich-g2) are submitted one by one and one of the clusters is shut down when the jobs are running. Since MPICH-G2 does not check the resource availability, MPICH-G2 still dispatches job to all four clusters. Moreover, the round-robin assignment of MPICH-G2 dispatches the parallel jobs as evenly distributed as possible. Hence, after one cluster becomes unavailable, most of the subsequent jobs fail. In VJM, with the pre-check mechanism, the subsequent jobs will be dispatched to other available clusters, and those jobs queued for the offline cluster(s) are migrated to other clusters by run-time cluster availability check.

4.1.2 Co-allocation of cross-domain resources and implications for virtual organizations

Resource allocation deadlock and unavailability can both result in the failure of cross-domain parallel jobs. To verify the VJM's dependability and performance for cross-domain parallel jobs, we deployed a Gfarm data grid environment consisted of 3 clusters (Table 1). The test application is *mpiblast-g2*, the MPI version of the bioinformatics application BLAST, compiled with MPICH-G2. Three test cases simulate the grid status under idle (all nodes are idle), moderate (requested resource less than number of idle nodes) and busy (requested resource more than number of idle nodes) conditions. The jobs are submitted using either mpich-g2 or CSF4, with increasing number of CPU's. Holder jobs which run a simple 'sleep 30' are submitted periodically to simulate jobs submitted locally, resulting in a queue length of 2 on one or two of the clusters.

The average amount of time to obtain resource co-allocation time (when all the requested nodes are available and the job starts to execute) is shown in Figure 4. Since *mpirun* dispatch jobs based on static resource configuration files, it is not aware of the actual number of nodes available within a cluster. When all the nodes are idling, mpirun is faster than VJM because of lower overhead (Case 1), whereas VJM takes

longer as the number of CPU's required increases. However, under moderate to busy conditions, since mpich-g2 distributes the subjobs equally to different hosts, regardless of node availability, the co-allocation time is significantly longer than the VJM (Case 2 and 3). However, under busy conditions, obtaining the co-allocation for the VJM is about the same as mpich-g2 (Case 3, 8 CPU) because VJmgr has to wait for some prior jobs to finish. However, since VJmgr can use backfilling of small jobs to nodes already allocated, actual resource usage would be much better than mpich-g2.

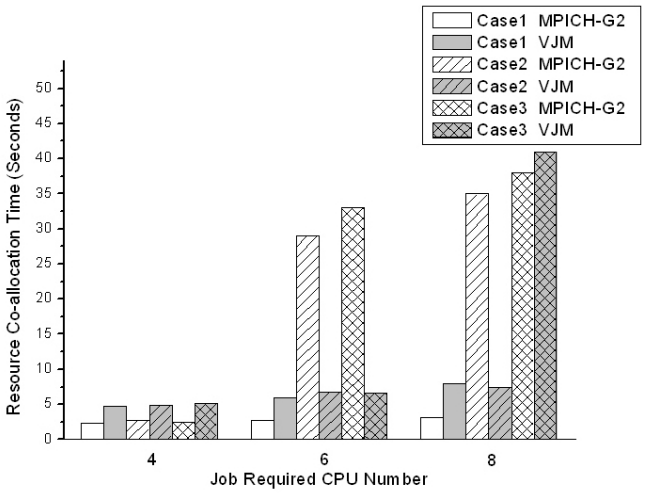


Figure 4. VJM improves the co-allocation efficiency of cross-cluster jobs over MPICH-G2 alone.

Table 1. Experimental setup for the evaluation of co-allocation time

Experiment	Case 1			Case 2			Case 3		
Grid Status	Idle			Moderate			Busy		
Cluster	A	B	C	A	B	C	A	B	C
Total CPU Number	3	3	4	3	3	4	3	3	4
Average Local Queue Length	0	0	0	0	2	0	0	2	2
Average Number of Available Hosts Reported by MDS	3	3	4	3	1	4	3	1	2
Total Available Hosts	10			8			6		
Required CPU Number for Job	4, 6, 8			4, 6, 8			4, 6, 8		

4.1.3 Deadlock detection and resolution

When required resources of all pending submitted jobs at same time exceed the total of available resource, resource deadlock may occur. In a grid environment composed of 2 clusters A and B, each with 3 compute nodes, two accounts, user1 and user2, are set up. On cluster A, user1's priority is higher than user2's, while on cluster B, user2's priority is higher than user1's. We then submit a parallel job which requires 6 machines by user1 and user2 at same time. With MPICH-G2, neither of the two jobs can start. Moreover, the resource deadlock results in the two clusters becoming unavailable. On the other hand, with VJM, if resource deadlock occurs, the task with lower priority on the meta-scheduler will be killed and re-allocated, the resource deadlock is broken.

The more resources a job requires, and the more jobs are involved in resource competition at same time, the more chances for resource allocation deadlock. In the following case, 100 random parallel jobs are generated, all of which require more than 1 cluster to run. Two clusters are used to submit jobs simultaneously. With MPICH-G2, the incidence of resource deadlock is 7 times out of 100 jobs. Moreover,

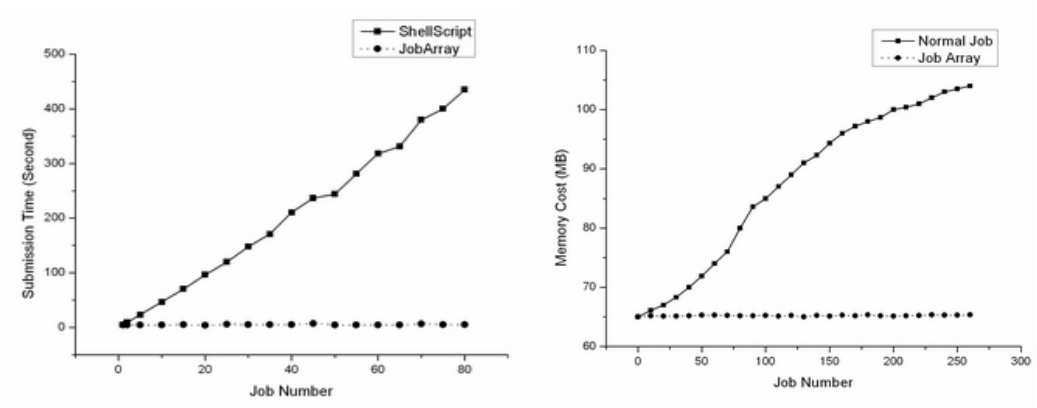
the resource deadlock results in the failure of all subsequent jobs. On the other hand, with VJM, if resource deadlock occurs, it is detected and resolved.

4.1.4 Back-filling

To verify small jobs backfilling, small jobs were inserted randomly, whose durations last less than 10 seconds, by different users, during the parallel job submission, With Gfarm, the backfilled jobs are completed successfully.

4.2 Evaluation of Array Job Plug-in

The array job plug-in is designed for the application which is composed of many pleasantly parallel jobs, like AutoDock and Blast. Using Blast as an example, the job submission time and memory cost of an array job are investigated. To ensure that the jobs are able to access their required data on any machines, we setup Gfarm on all clusters to provide a global file system. Figure 5-1 shows the comparison of job submission time using a shell script and a job array. With the CSF4 command line interface, the submission time of Blast jobs by a shell script is almost linear with respect to the number of jobs, since each invocation incurs an overhead .On the other hand, the submission time using a job array is nearly same regardless of the number of jobs.



5-1 Job Submission Time 5-2 Memory Cost

Figure 5 Shell Script and Job Array Submission Time

Figure 5-2 shows the comparison of the total memory costs of multiple normal jobs (jobs submitted via using globusrun) and the memory cost of job arrays made up of multiple subjobs. The job array functionality is implemented in CSF4 hosted in GT4 (Globus Toolkit 4 [17], [18]). The base memory cost is about 60MB. For normal job, the average memory cost for one job is 0.18MB, with the total memory increasing almost linearly. For job array, only dispatched jobs are kept in memory, so the memory footprint is very small.

4.3 Distributed Array Job for Blast

The Job Array feature is suited for applications like AutoDock or Blast. Previously, end users need to write a shell script to submit all the jobs one by one, causing a lot of SOAP communication overhead. By naming the input/output files in a particular format, with a variable number *i* from 1 to N, an arbitrary integer representing the total number of jobs, a user submits the job only once, the metascheduler generates all the subjobs automatically. For example, the user submits a job, myArrayJob, with arrays size n, myProg as the binary executable, and myInput/myOutput as input/output. The metascheduler generates N subjobs, myArrayJob[1]…myArrayJob[N]. Each sub job myArrayJob[*i*] (*i*=1..N) executes same binary, myProg, takes myInput.*i* as the input file name, and puts the output in myOutPut.*i* file.

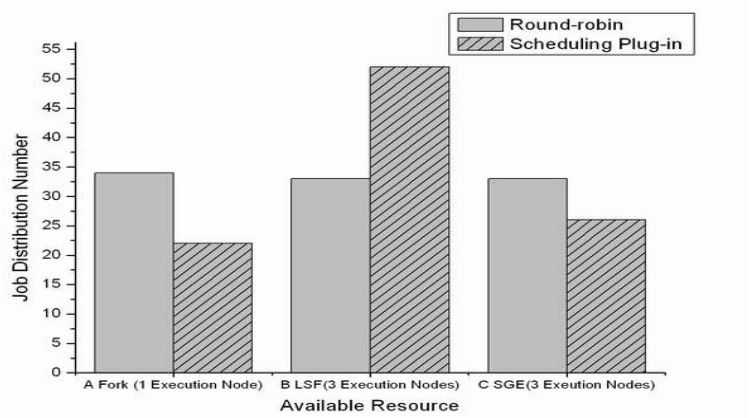


Figure 5. Job Distribution Map by Round-robin and Array Job Plug-in to 3 different clusters.

To test the performance of array job scheduling policy, we configured 3 available resources A, B and C as a Gfarm filesystem. A is a single machine; B is a local LSF (Load Sharing Facility) 6.0 cluster with 3 compute nodes; C is a remote SGE (Sun Grid Engine) 6.0 cluster with 3 compute nodes. On cluster C, we submit a holder job ('sleep') to the local scheduler to compute nodes periodically. Blast is used as the test application, with 100 single sequence similarity searches performing reading and writing to the Gfarm filesystem. The jobs are scheduled by Round-robin policy plug-in and Array Job plug-in respectively. The experiment is done twice using the ecoli.nt (5 MB) nucleotide database from NCBI.

Figure 5 depicts the job distribution map of the two policies. The Round-robin algorithm dispatches jobs to three resources evenly regardless of the capability and workload of clusters. The array job plug-in dispatches jobs according to unfinished subjob numbers. Since cluster B is a local cluster and more capable, it can complete jobs faster, so the plug-in dispatches more jobs to it. On Cluster C, as a part of the resource is occupied by its local jobs, the time needed for all jobs to finish is longer. Although the metascheduler does not know the actual situations of the cluster, it can submit fewer jobs to cluster C. Table 2 shows the last job exit time of the subjobs scheduled to each cluster. With the array job plug-in, a better work load balancing is achieved. Hence, the last of the all the subjobs under the array job scheduling policy is finished in 12 min 21 s, while that under round robin is not completed till 24 min 5 s. Therefore, round robin is inefficient as it sends the same number of jobs to Cluster C, regardless of the quality of service, and delays the wall clock time for job completion. On the other hand, although cluster A receives fewer jobs under the array job plug-in policy, the wait time (scheduling interval) between each dispatch event actually causes the last subjob exit time to be delayed. This suggests that typical subjob runtime may be better correlated with the default scheduling interval as one of the means to achieve optimal local resource usage efficiency.

Table 2. Last job exit time for subjobs allocated to each cluster using two different scheduling policies. The total time taken for all jobs to complete using each policy is indicated in bold.

Cluster		A	B	C
Algorithms	Round-robin	7 min 15 s	5 min 38 s	**24 min 5 s**
	Array Job	10 min 8 s	10 min 42 s	**12 min 21 s**

5 Transparent Access and Improved Usability of the Grid

As more and more clusters as well as and grids of clusters or virtual organizations (VO) spring up, the ability to simplify the scheduling of tasks across VO becomes critical. The difficulty in bridging the gap between the grid services and the target user communities remains significant due to the unfriendly interface of grid software. While the high energy physics community with computational prowess has taken a lead in

the development of integrated systems for grid computing and data management, the biomedical research community requires additional effort to lower the barrier of entry by making the grid access transparent and easily accessible.

5.1 My WorkSphere

Typically end users in life sciences access computational resources through a web based interface to a popular application such as Blast available at the National Center for Biotechnology Information (NCBI), or through desktop applications that may solve small problems locally. With Moore's law continue to hold true by increasing the number of computing cores per processor, desktops may become more and more powerful. However, the pervasive availability of distributed grid resources and petascale facilities will continue to entice end users as they begin to tackle problems at ever increasing scales [19].

My WorkSphere [20] is an integrative environment which leverages open source components to provide easy access to grid computing resources. In this particular environment, a number of open source software packages are adopted and integrated, such as the GridSphere portal framework and Gridportlets [21], the GAMA server and portlet [22], the Opal web service toolkit [23], the community scheduler framework 4 (CSF4), Gfarm, Globus Toolkit and Commodity Grid Kits [24], and Rocks [25]. The purpose of My WorkSphere is to prototype an environment where users can easily gain access to grid computation resources; run jobs without worrying about what resources they are using, and deploy applications easily for use on the grid. It is one of the TeraGrid Science Gateways for the biomedical community researchers under development by the National Biomedical Computation Resource (NBCR, [26]), in collaboration with academic researchers and developers worldwide.

5.2 CSF4 Portlet

The CSF4 portlet v1.0 is developed through the collaboration between Jilin University and University of California, San Diego (UCSD) driven by the needs of My WorkSphere. It is a java based web application for dispatching jobs to a CSF4 metascheduler server through a web browser.

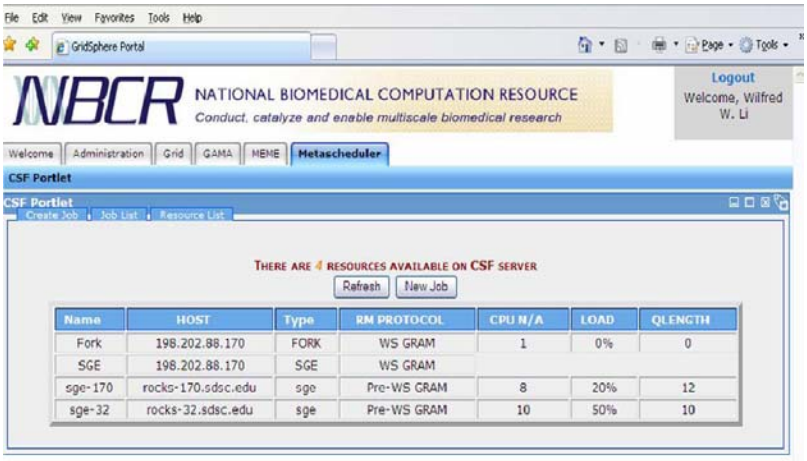


Figure 6 The CSF4 Portlet in My WorkSphere

The CSF4 portlet not only presents the functionality of CSF4 itself, but also provide a generic interface for users to submit jobs, view job specifications and job history, monitor job status, and get job output from remote sites using GridFTP [27]. Additionally, the CSF4 portlet uses the gridportlets to provide methods for tracking Globus Security Service (GSS) credentials for a given user either from GAMA server or directly from a MyProxy server [28]. Besides credential management, the APIs provided by GridSphere and the gridportlets are also used to implement account management services. The CSF4 portlet supports the

integration of CSF4 and Gfarm file system.

6 Conclusion and Future Work

This paper describes an extensible scheduler plug-in model for metascheduling using CSF4 with support for customized metascheduling policies for life sciences applications. A virtual job model is proposed for efficient and stable execution of cross-domain parallel jobs. The model introduces a virtual job mechanism and a grid-adaptive backfilling algorithm. The model can guarantee parallel job synchronous startup, break resource allocation dead lock and alleviate resource waste. This model is highly suitable for MPI applications compiled with MPICH-G2. In addition, a Job Array feature has been developed to demonstrate the power of the plug-in module, which supports popular pleasantly parallel applications such as AutoDock and Blast. Through international collaboration activities espoused under the Pacific Rim Applications and Middleware Assembly (PRAGMA, [29]), a CSF4 portlet has been developed to make possible use of the grid in a 'submit once and run anywhere' style. Through projects like My WorkSphere, and the PRAGMA Biosciences Portal, the open source computational data grid computing platform may be used, and improved with feedback from an international user base.

We are currently using the new features implemented in this paper with AutoDock and mpiBLAST-G2 [30] in collaboration with biomedical researchers in different projects, including the Community Cyberinfrastructure for Advanced Marine Microbial Ecology Research and Analysis (CAMERA, [31]). The scale of these research activities precludes the inclusion of these data in the current manuscript, and will be reported elsewhere when ready. From an infrastructure perspective, we are working towards the following goals: 1) Use of CSF4 as a metascheduler for Opal based web services, with Gfarm as an data and application repository; 2) Support data uploads from client side, either from CSF portlet or an Opal based web service client; 3) Improve the scheduling plug-in framework and implement more scheduling policies as plug-ins 4) Deploy the system for production use; 5) As a TeraGrid Science Gateways portal, we are developing ways to interoperate with the TeraGrid [32], as well as other VO's within the Open Science Grid (OSG, [33]).

7 Acknowledgement

The work of the paper is supported by Jilin University under Grant No.419070200053 and Grant No.420010302338, CNSF under Grant No.60473099 and NSF of Jilin Province under Grant No. 20060532 and No.20040119. P.W. Arzberger, W.W. Li wish to acknowledge PRAGMA as supported by NSF Grant No. INT-0216895 and INT-0314015; NBCR as supported by NIH NCRR P41 RR08605; CAMERA project as supported by the Moore Foundation.

8 Reference

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, *PVM:Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing*. Cambridge, MA: MIT Press, 1994.

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard.," *Parallel Computing*, vol. 22, pp. 789-828, 1996.

[3] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface," *J. Parallel Distrib Comput*, vol. 63, pp. 551-563, 2003.

[4] X. Wei, Z. Ding, S. Yuan, C. Hou, and H. Li, "CSF4: A WSRF Compliant Meta-Scheduler," presented at International Conference 06' on Grid Computing and Applications, Las Vegas, USA., 2006.

[5] D. S. Goodsell, G. M. Morris, and A. J. Olson, "Automated docking of flexible ligands: applications of AutoDock," *J Mol Recognit*, vol. 9, pp. 1-5, 1996.

[6] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," *Nucleic Acids Res*, vol. 25, pp. 3389-402, 1997.

[7] W. W. Li, S. Krishnan, K. Mueller, K. Ichikawa, S. Date, S. Dallakyan, M. Sanner, C. Misleh, Z. Ding, X. Wei, O. Tatebe, and P. W. Arzberger, "Building cyberinfrastructure for bioinformatics using service oriented architecture," presented at Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops Singapore, 2006.

[8] Silver, "Moab grid scheduler," 2000, http://www.supercluster.org/projects/silver/index.html.

[9] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," presented at Intl. Workshop on Quality of Service, University College London, UK, 1999.

[10] R. S. M. I. M. L. Eduardo Huedo, "A framework for adaptive execution in grids," *Software: Practice and Experience*, vol. 34, pp. 631-651, 2004.

[11] J. Frey, T. Tannenbaum, M. Livny, T. I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, vol. 5, pp. 237-246, 2002.

[12] D. Abramson, J. Giddy, and L. Kotler, "High performance parametric modeling with Nimrod/G: Killer application for the global grid?," presented at IPDPS, 2000.

[13] K. Czajkowski, I. Foster, C. Kesselman, C. Sander, and S. Tuecke, "SNAP: A Protocol for negotiating service level agreements and coordinating resource management in distributed systems.," *Lecture Notes In Computer Science*, vol. 2537, pp. 153-183, 2002.

[14] M. Massie, B. Chun, and D. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation and Experience," *Parallel Computing*, vol. 30, pp. 817-840, 2004.

[15] P. Uthayopas, J. Maneesilp, and P. Ingongnam, "SCMS: An Integrated Cluster Management Tool for Beowulf Cluster System," presented at PDPTA, Las Vegas, Nevda, USA, 2000.

[16] O. Tatebe, N. Soda, Y. Morita, S. Matsuoka, and S. Sekiguchi, "Gfarm v2: A Grid file system that supports high-performance distributed and parallel data computing," presented at 2004 Computing in High Energy and Nuclear Physics, Interlaken, Switzerland, 2004.

[17] I. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems," *Lecture Notes In Computer Science*, vol. 3779, pp. 2-13, 2006.

[18] Globus, "The Globus Alliance," 2004, http://www.globus.org.

[19] W. W. Li, N. A. Baker, K. Baldridge, J. A. McCammon, M. H. Ellisman, A. Gupta, M. J. Holst, A. D. McCulloch, A. Michailova, P. Papadopoulos, A. Olson, M. Sanner, and P. W. Arzberger, "National Biomedical Computation Resource (NBCR): Devloping End-to-End Cyberinfrastructure for Multiscale Modeling in Biomedical Research " in *CTWatch Quarterly*, vol. 2, 2006, pp. 6-17.

[20] Z. Ding, Y. Luo, X. Wei, C. Misleh, W. W. Li, P. W. Arzberger, and O. Tatebe, "My WorkSphere: Integrated and Transparent Acess to Gfarm Computational Data Grid through GridSphere Portal with Metascheduler CSF4," presented at 3rd International Life Sciences Grid Workshop, Yokohama, Japan, 2006.

[21] J. Novotny, M. Russell, and O. Wehrens, "GridSphere: a protal framework for building collaborations," *Concurrency and Computation: Practice and Experience*, vol. 16, pp. 503-513, 2004.

[22] K. Bhatia, S. Chandra, and K. Mueller, "GAMA: Grid Account Management Architecture," presented at 1st IEEE International Conference on e- Science and Grid Computing, Melbourne, Australia, 2006.

[23] S. Krishnan, B. Stearn, K. Bhatia, K. Baldridge, W. W. Li, and P. W. Arzberger, "Opal: Simple Web Service Wrappers for Scientific Applications," presented at International Conference for Web Services, 2006.

[24] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and*

Computation: Practice and Experience, vol. 13, pp. 643-662, 2001.
[25] ROCKS, "Rocks Cluster Distribution," 2005, http://www.rocksclusters.org.
[26] NBCR, "National Biomedical Computation Resource," 2005, http://nbcr.net.
[27] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, "GridFTP: Protocol Extension to FTP for the Grid, Grid Forum Internet-Draft," 2001.
[28] J. Novotny, S. Tuecke, and V. Welch, "An Online Credential Repository for the Grid: MyProxy," presented at High Performance Distributed Computing (HPDC), 2001.
[29] PRAGMA, "Pacific Rim Applications and Grid Middleware Assembly," 2004, http://www.pragma-grid.net/.
[30] A. Darling, "The Design, Implementation, and Evaluation of mpiBLAST,," presented at ClusterWorld, San Jose, 2003.
[31] CAMERA, "Community Cyberinfrastructure for Advanced Marine Microbial Ecology Research and Analysis," 2006, http://camera.calit2.net/.
[32] TeraGrid, "TeraGrid," 2004, http://www.teragrid.org.
[33] OSG, "Open Science Grid," 2006, http://www.opensciencegrid.org/.