“Gridifying” Aerodynamic Design Problem Using GridRPC

Quoc-Thuan Ho, Yew-Soon Ong, and Wentong Cai*

Nanyang Technological University
School of Computer Engineering
Nanyang Drive, Singapore 639798

Abstract. This paper presents a “gridifying” process for aerodynamic wing design as a case study of complex engineering design problems. In order to assist engineers and scientists to solve the problems on the Grid environment effectively, we developed an API based on GridRPC, a Remote Procedure Call standard interface for Grid-enabled applications. In this work, we also provide mechanisms to reduce communication cost and schedule GridRPC requests across multiple clusters on the Grid. These facilities are incorporated in our GridRPC API implementation.

Keywords: Aerodynamic design; Grid computing; GridRPC; Composite service; Metascheduling

1 Introduction

Thanks to the advent of more and more powerful computers and progress of discretization methods and iterative solution algorithms, paper-based engineering design has been transformed towards 3D solid models and computer simulations. The increasing use of computational power in the design process helps design engineers reduce the design cycle time, save costs, and at the same time assures the quality of the products. These are the desirable goals in the industries, especially in aircraft manufacture industry which is increasingly exposed to considerable commercial pressures.

Figure 1 shows a typical life cycle of the complex engineering design. The design cycle begins with Computer Aided Design (CAD) modelling, followed by pre-processing, optimization, and visualization phases. It is common in the present complex engineering design life cycle that the preprocessing and optimization take up the overwhelming bulk of the computation. As a result, bottleneck often happens at these phases. One way to mitigate the problem is to employ parallelism in the design process via multiple computing nodes.

Grid computing offers very powerful and inexpensive computational resources that can be pooled together to solve large problems. Recently, many computationally intensive applications in complex engineering design have been deployed successfully on the Grid. Employing Grid computing can help reduce the bottleneck problem in preprocessing and optimization phases by using clusters of

* Contact Author: email – aswtcai@ntu.edu.sg

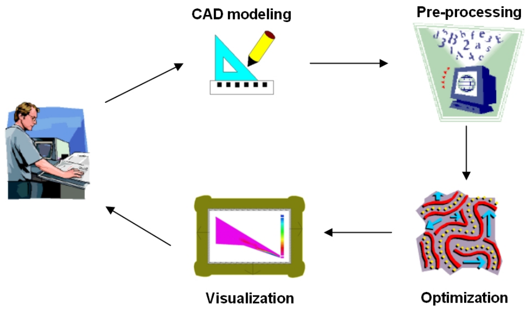

Fig. 1. Design Cycle for Complex Engineering Problems

computers to analyze and optimize different alternative designs in parallel . However, “gridifying” an application involves coping with many challenging issues, for examples, programming cost, interoperability, resource discovery and selection, as well as attaining high performance.

Applications can be “gridified” using Globus API, MPICH/G2, CoG, etc. [1]. We propose an implementation of GridRPC API to “gridify” our application since GridRPC API is relatively simple, well-understood, and offers high-level abstraction whereby complexity of Grid environment are hidden from the users [2]. Hence, using GridRPC can help reduce programming cost in “gridifying” or developing Grid-enabled applications. A successful example of using GridRPC can be found in [3]. Recently, GridRPC has also been implemented in NetSolve [4] and Ninf-G [5].

However, these current state-of-the-art GridRPC implementations lack the support for automatic resource discovery and selection on the Grid environment. As a result, users have to perform look-up and select resources that suit their needs manually before they can assign computing tasks to the respective resources. This leads to inefficiency since a large number of resources exist on the Grid and these resources are often dynamic. In addition, most optimization problems, for example, aerodynamic design problems, have a large number of evaluation tasks that requires many identical computations with different analysis parameter sets. Hence it is inefficient if interactions between clients and resources are repeated many times for the same remote procedure call. On the Grid, where resources are distributed on Wide Area Network (WAN) that has high communication latency, performance of applications may decrease drastically.

In this paper, we present a GridRPC API implementation and the “gridifying” process that takes into account the above issues. Further, automatic resource discovery and selection are considered as basic functionalities of our GridRPC API implementation. In particular, a mechanism called *composite service* is introduced as an extension of the GridRPC API. It is employed as a means to reduce communication cost between clients and resources. In addition, a metascheduler that schedules GridRPC requests and balances workload across multiple clusters on the Grid is also introduced.

The rest of this paper is organized as follows. Section 2 briefly describes a general aerodynamic wing design problem and previous work deployed on NetSolve platform. In section 3, we present a "gridifying" process using the GridRPC API we have developed. Discussion on composite service and the metascheduler for load balancing in Grid environment are also presented in this section. Experimental results on aerodynamic wing design problem using the GridRPC API developed are discussed in section 4. Finally, section 5 draws conclusions and outlines future work.

2 Aerodynamic Wing Design Problem

Aerodynamic design, one of the most frequently tackled problems in aeronautics, is a computationally intensive application. Due to the large design spaces, usually stochastic optimization algorithms such as Genetic Algorithms (GA) are employed in the aerodynamic search in order to arrive at a near optimum design [6, 7]. GAs typically require thousands of function evaluations and hence require excessive CPU time to locate a near optimal solution, where a single function evaluation involving high-fidelity analysis codes could take up to minutes and hours of computation time. Hence, it is not uncommon that the design of an aircraft wing using evolutionary algorithms may take up several months of CPU time even on a supercomputer. This often poses a serious impediment to the practical application of evolutionary optimization to complex engineering design problems as the design process may prove to be computationally intractable if an optimal design is desired.

Since GAs are population-based technique, parallelizing the algorithm is easily achieved such that several designs in each generation can be evaluated simultaneously across the available machines. In a previous work [8], the problem was tackled by using a parallelized evolutionary algorithm for computationally expensive problems deployed on NetSolve platform [9]. Analysis codes are wrapped and deployed on multiple NetSolve servers. Then, a large number of evaluation tasks are farmed to NetSolve servers via the NetSolve agent. As a result, improvement of the design cycle has been attained.

Presently, engineers and scientists still desire to shorten the design cycle time and at the same time assure design quality. However, due to large search spaces in complex engineering design problems, in particular, in multidisciplinary design optimization, computational budget based on a single cluster may be limited. Therefore, migrating the most time consuming part of the design cycle to a Grid environment which has limitless computational resources will help shorten the entire design cycle time. In the subsequent section we present the efforts required to deploy an aerodynamic wing design problem onto the Grid.

3 "Gridifying" Aerodynamic Wing Design Problem Using GridRPC

3.1 Infrastructure for GridRPC

The GridRPC API was built on top of Globus [10] that represents the defacto standard of Grid infrastructure and enables interoperability among various Grid resources. Many services provided by Globus such as Globus Monitoring and Discovery Service (MDS), Grid Resource Allocation Manager (GRAM), Grid Resource Information Service (GRIS), Grid Security Infrastructure (GSI), Global Access to Secondary Storage (GASS), and GridFTP were employed in our GridRPC API implementation. In particular, we consider resource discovery, selection and scheduling as basic functionalities in our implementation. They were incorporated in the implementation so that GridRPC clients can interact with Grid resources autonomically.

It is difficult for a GridRPC client program to know how many resources exist and where they are. Thus, the client needs a centralized like directory service to locate resources. Here, Lightweight Directory Access Protocol (LDAP) [11] is employed to query the MDS database for the resource and workload information. The GridRPC initialization function requires a simple entry for MDS server and a virtual organization name to retrieve the available resources and check whether or not these resources are capable of servicing further GridRPC requests. Checks are done for software (remote procedure and specialized libraries), hardware (platform, operating system, etc.), and the workload. In order to monitor and provide information of clusters of computer, the Ganglia monitor toolkit [12] is employed. All information is published on the central MDS server since Globus MDS schema can be extended to include many types of information, for example, the dynamic information provided by Ganglia.

3.2 Composite Services

Execution of a GridRPC request is done at the resources. When a resource is not a single computer, e.g. a clusters of computers, the remote invocation of a procedure is usually carried out using resource managers such as Condor [13] , PBS [14] , SGE [15], etc. All client requests must be submitted through the resource manager. Subsequently, the resource manager schedules these requests across several computing servers and then harvests the results obtained for the clients. The remote procedure calls are considered here as basic services. In an application that requires a large number of identical basic services with different parameter sets, invoking these basic services individually may prove to be very inefficient. This is because the interactions between clients and resources would be intensively high. In particular, since Grid resources are geographically distributed across the Internet, performance of applications may decreases drastically due to the high communication latency involved.

In order to avoid the above mentioned disadvantages, we extend GridRPC API for composite services. An additional parameter that indicates the number of basic services will be carried out inside a composite service is introduced in GridRPC API. When the number is equal to one, the composite service becomes a basic service. Using composite service, a set of basic GridRPC requests can be bundled into one composite GridRPC request such that the amount of communications between clients and resources is reduced drastically. The complexity of communication is also hidden from the user.

In this work, we employed NetSolve as the middleware to implement composite services. These services are NetSolve programs that are also GridRPC-enabled. However, since NetSolve does not explicitly attempt to balance the workload across computing nodes [9], it cannot avoid overloading situation whereby a certain computing node is overloaded by multiple uncooperative clients. When a client sends multiple asynchronous requests within a short time, the NetSolve agent may fail to obtain up-to-date workload information on the computing nodes and thus provide inaccurate prediction. To tackle this issue, we modified the scheduler of NetSolve agent such that parallel tasks can be distributed fairly across different computing nodes.

Figure 2 presents the execution mechanism for the GridRPC systems. It is assumed that applications are developed using our extended GridRPC API. At the front-end, the GridRPC client activates several remote procedure calls. The metascheduler embedded in GridRPC API implementation then farms the requests to the available resources based on workload and resource information recorded in the Globus MDS. Hence, every time a new resource joins the Grid, it will need to register to the Globus MDS. In addition, its workload information obtained from Ganglia is pushed onto Globus MDS. At the back-end, once the GRAM's gatekeeper of a cluster receives a composite service request, an instance of the composite service will be initialized on the master node of the cluster. Subsequently, the set of basic service requests will be farmed across multiple computing nodes in the cluster. The responsibility of the NetSolve agent is to perform local scheduling and resource discovery across computing nodes in the cluster for basic services.

3.3 Metascheduling

Resource selection and scheduling plays a critical factor on the performance of Grid applications. In order to offer effective and efficient schedules of GridRPC programs, the scheduler should possess ability to estimate the execution time of services on resources as well as workload increment of the resources during program execution. Scheduling mechanism has been incorporated into GridRPC API implementation such that each client has its own metascheduler. Upon initialization of a GridRPC-enabled client, the metascheduler obtains a list of available resources together with their status from Globus MDS. Whenever the GridRPC client makes a GridRPC call, the metascheduler selects the most suitable resource and submits the request to the resource.

The metascheduler not only performs match-making between the request's requirements and available resources but also predicts workload change of the resources once a request is submitted. Prediction is required since retrieving

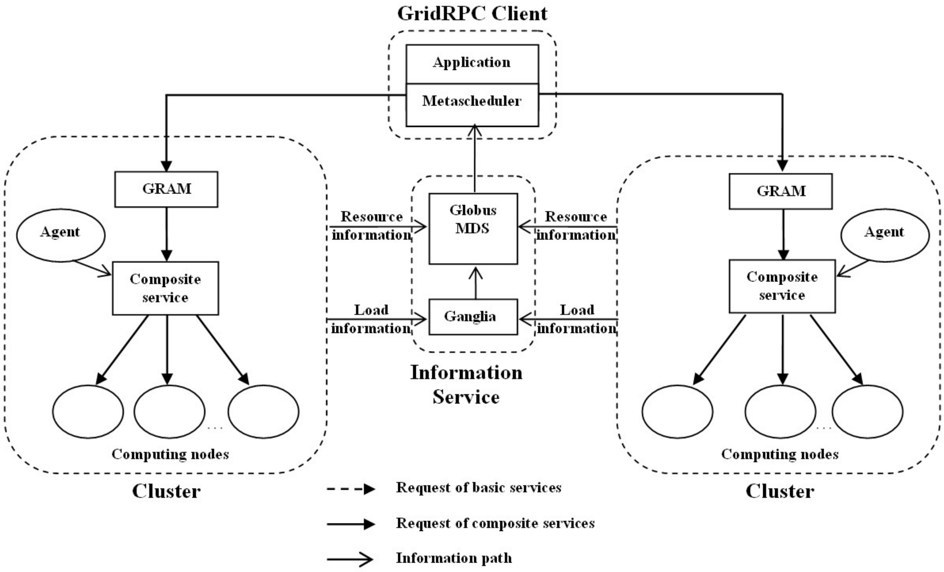

Fig. 2. GridRPC execution system

resource status from the resource monitor is time consuming and the information may not be up-to-date due to network latency. Especially, when asynchronous requests are produced in a loop, resource status cannot be updated as fast as the requests generated. Since the metascheduler is incorporated in the GridRPC API implementation and it schedules the requests online, the scheduling algorithm should not be complex. Hence, we use a heuristic algorithm with $O(nmlogm)$ complexity to schedule GridRPC requests (see Figure 3). Notes that static scheduling for n requests across m resources is NP-complete and an exhaustive search is of $O(m^n)$ complexity. It is too expensive, especially when the number of available resources is large.

It is difficult to obtain a the normalized formula to evaluate the performance capabilities of diverse platforms. Therefore, benchmarking the resource using benchmark libraries such as LINPACK, SCALAPACK may be a good way to obtain this kind of information. In the algorithm, $Raw\_Performance$ represents the performance of a resource (in MFLOPS) and is obtained by benchmarking using an analysis code.

Computing cost of a remote procedure denoted by $TaskSize$ variable depends much on the application. It can be provided by the user. In order to have an accurate estimation of the $TaskSize$ value, users can benchmark their library that is executed by the remote procedure call and publish $TaskSize$ value as part of the remote procedure call information in Globus MDS. The $TaskSize$ value should preferably be platform-independent. Specifically, the number of floating point operations required to finish the computation can be used in this case.

```
Contact the resource monitor (Globus MDS)
Get m available resources R={R_0, R_1, ..., R_m}
Get W={W_1, W_2, ..., W_m} as workload values of the above resources
repea
    Receive a request r from the application
    Set L = ∅, L is a list of candidate resources for r
    for (each resource R_i in R)cdo
        if R_i can serve r
            Calculate F_cost(R_i, r) as computation time for request r on resource R_i
            Put R_i in L
        endif
    endfor
    Sort L by increasing cost
    Submit r to R_k, R_k is the first resource in L
    Update W_k = W_k + F_incWorkload(R_k, r)
until (program terminates)
```

Fig. 3. Metascheduling algorithm

The average number of processes in the ready queue waiting to be executed over the last minute, i.e., the "*load_one*" metric, can be used as a reasonable indicator of the dynamic loading status of a system. This metric can be obtained from Ganglia. If the resource is a cluster, the average "*load_one*" values is employed. In Figure 3, the workload value of resource R_i, W_i, is initialized using this metric.

Based on the workload, raw performance of the resource, and task size of a request, computation time for processing the request r on resource R_i can be estimated as below:

$$F_{cost}(R_i, r) = \frac{W_i \times TaskSize(r)}{Raw\_Performance(R_i)}$$

After the request is submitted to the resource, we estimate future workload of the resource for scheduling further request.

$$W_i = W_i + F_{incWorkload}(R_i, r)$$

Intuitively, workload increment value is directly proportional to the $TaskSize$ of the remote procedure. When the remote procedure is computationally intensive, it consumes more processor time and the workload increment is thus equal to one since the procedure is likely to take some time to complete. Therefore, the workload increment, $F_{incWorkload}(R_i, r)$, is defined as follows:

$$F_{incWorkload}(R_i, r) = \begin{cases} \frac{TaskSize(r)}{Raw\_Performance(R_i)}, & \text{if } \frac{TaskSize(r)}{Raw\_Performance(R_i)} < 1.0; \\ 1.0, & \text{otherwise.} \end{cases}$$

4 Experimental Results

In this experiment, the design of airfoil for minimum drag was deployed on 3 clusters within the Nanyang Technological University Campus Grid. Their configuration and initial load are presented in Table 1. Each cluster has 10 computing nodes and 1 master node. Here, 200 potential airfoil designs are analyzed in parallel. The analysis codes employed here are reduced version of 2D Euler codes. In our estimation, the $TaskSize$ value for one analysis task is approximately 14,062,632,000 floating point operations.

We used our extended GridRPC API with composite services in our implementation. In the GridRPC client program, 200 evaluation tasks are submitted in a loop of 20 asynchronous calls, each of which invokes a composite service. Each instance of the composite service analyzes 10 designs on a cluster. Instead of using the basic service that needs 200 remote procedure calls, we employed composite service with only 20 calls. Hence, the communication cost between the GridRPC client and the resources is reduced significantly. We also minimized communication cost by replicating the design parameters on three clusters.

Table 1. Properties of three clusters

	HPC Cluster	HPC2 Cluster	Sun Cluster
Raw performance	480.943 MFLOPS	90.147 MFLOPS	85.833 MFLOPS
Initial workload	3.188	0.020	0.021

Upon receiving the analysis jobs submitted by the GridRPC client, the metascheduler distributed 13, 4, and 3 requests to HPC Cluster, HPC2 Cluster and Sun Cluster, respectively. The workload of three clusters servicing these service requests are presented in Figure 4. At the beginning of the experiment, HPC Cluster had the highest workload because of heavy local tasks. However, since the metascheduler takes into account both workload and raw performance on scheduling, it distributed more requests to HPC Cluster because it is the most powerful system. From the figure, it can be seen that although 13 requests were assigned to the HPC cluster, the load-one index did not increase much compared with the other two clusters. In contrast to HPC Cluster, Sun Cluster had the lowest performance since its average clock speed is the lowest. Therefore, it will take more time to finish a small number of composite requests.

Load information shown in Figure 4 also implicitly presents the differences among computing platforms, for instances, between HPC2 Cluster, an Intel platform with Linux operating system and Sun Cluster, a Sun platform with Solaris operating system. Although both clusters started with the similar load and their raw performance values were approximately equal, the peak workload values and their changes looked different.

On the contrary, load balancing within a cluster is easier. A snapshot of workload of computing nodes inside HPC2 Cluster during execution grabbed from Ganglia Web monitoring page is shown in Figure 5. In this snapshot, 40 designs were analyzed from 18:40 to 19:00. Workload of the master node (*ec-pdccm*

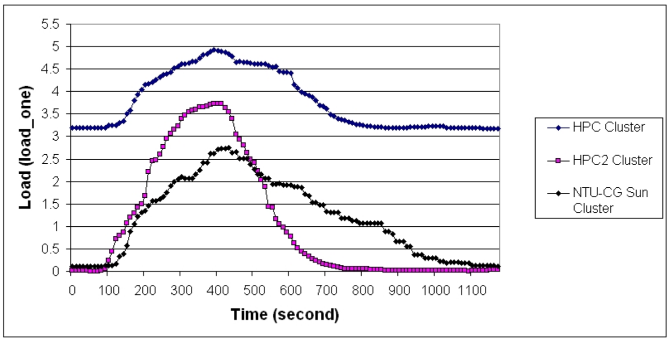

Fig. 4. Workload of three clusters

machine) differed from that of the computing nodes because it only performed scheduling computing tasks of composite services.

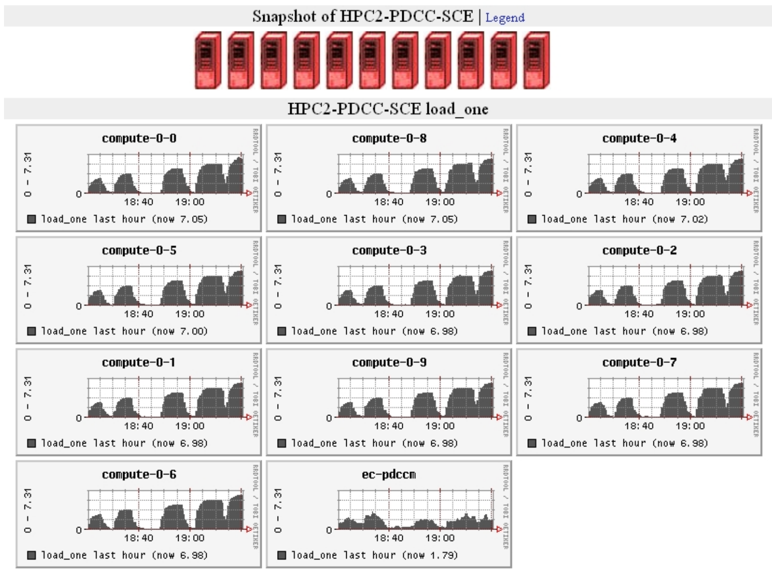

Fig. 5. Load characteristics of a cluster during optimization phase

5 Conclusions and future work

"Gridifying" an application over the Grid is still hindered by a lack of suitable middleware. In this paper, we present a design and implementation of the

GridRPC API to attain a seamless and transparent access to Grid resources. How to "gridify" a problem in aerodynamic wing design using our GridRPC implementation is also discussed. Composite services in our implementation can hide complexity of underlying execution across multiple clusters of computers and reduce communication cost between GridRPC clients and Grid resources. Experimental results indicate that the metascheduler can successfully select the suitable resources for its requests according to the resource status and balance the workload among different heterogeneous resources of diverse capabilities. In next step, we will look at dynamic approach to bundle GridRPC requests. The scheduling thus can be more adaptive to clusters of different sizes and workloads across multiple resources.

However, since each client is equipped with its own metascheduler, a problem may be exposed when several GridRPC clients submit jobs at the same time. In this scenario, "selecting the best resource" strategy used in the current implementation may prove to be inefficient because of resource competition. Our future work will also investigate involve investigating centralized and optimistic approaches for scheduling. In the former case, all clients will submit jobs through a centralized scheduler. When all of clients are decentralized, optimistic mechanism can be employed to improve the prediction of workload.

Reusing existing modules and libraries are desirable when applications become larger and complicated. Nested GridRPC may prove to be a good form to support service providers constructing complex functional services. A composite service presented in this paper can be seen as a form of nested GridRPC of two levels, one from the GridRPC client to a cluster of computers and the other from the master node running composite procedure to computing nodes of the cluster. Thus, the current implementation needs to be further extended to handle multi-level nested GridRPCs.

References

1. C. Lee, S. Matsuoka, D. Talia, M. Mueller, G. Allen, J. Saltz. A Grid Programming Primer. Advanced Programming Models Research Group, Global Grid Forum. Available at: `http://www.eece.unm.edu/~apm/docs/APM_Primer_0801.pdf`
2. H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. GridRPC: A Remote Procedure Call API for Grid Computing. Advanced Programming Models Research Group, The Global Grid Forum, July 2002. Available at: `http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf`
3. H. Takemiya, K. Shudo, Y. Tanaka, S. Sekiguchi. Development of Grid Applications on Standard Grid Middleware. In *Workshop on Grid Applications and Programming Tools*, GGF8, June 25, 2003
4. S. Vadhiyar, K. Seymour, and M. Miller. Implementation of GridRPC over NetSolve. In *GGF5 APM Meeting*, Edinburgh, Scotland, July 22, 2002.
5. H. Nakada, Y. Tanaka, S. Matsuoka, and S. Sekiguchi. Ninf-G: A GridRPC System on the Globus Toolkit. In *Grid computing: Making the Global Infrastructure a Reality*, John Wiley & Sons, March 2003.

6. K. Yamamoto, O. Inoue. Applications of Genetic Algorithm to Aerodynamic Shape Optimization. In *Proceedings of the 12th AIAA Computational Fluid Dynamics Conference*, 1995.
7. D. Quagliarella, J. Periaux, C. Poloni, G. Winter (Eds.). Genetic Algorithms and Evolution Strategies in Engineering and Computer Science. John Wiley & Sons, Chichester, 1998.
8. Y. S. Ong, P. B. Nair, and A. J. Keane. Evolutionary Optimization of Computationally Expensive Problems via Surrogate Modeling. In *American Institute of Aeronautics and Astronautics Journal*, 40(4), pp. 687-696, 2003.
9. D. Arnold, H. Casanova, and J. Dongara. Innovations of the NetSolve Grid Computing System. In *Concurrency: Practical and Experience 14(13-15): 1457-1479 (2002)*, 2002.
10. I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop, pp 4..18*, 1998.
11. I. Foster and G. von Laszewski. Usage of LDAP in Globus. Available at: `ftp://ftp.globus.org/pub/globus/papers/ldap_in_globus.pdf`
12. M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. February 2003.
13. D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In *Grid Computing: Making The Global Infrastructure a Reality*, John Wiley & Sons, March 2003.
14. PBS Documentation Set. Available at: `http://www.pbspro.com`
15. Grid Engine Documentations. Available at: `http://gridengine.sunsource.net/project/gridengine/documentation.html`