Faucets: Efficient Resource Allocation on the Computational Grid

Laxmikant V. Kalé, Sameer Kumar, Mani Potnuru,
Jayant DeSouza, Sindhura Bandhakavi
Department of Computer Science
University of Illinois at Urbana-Champaign
{kale,skumar2,potnuru,jdesouza,bandhaka}@cs.uiuc.edu

Abstract

The idea of a "Computational Grid" suggests that high end computational power can be thought of as a utility, similar to electricity or water. Making this metaphor work requires a sophisticated "power distribution" infrastructure. In this paper, we present the Faucets framework that aims at providing (a) user-friendly compute power distribution across the grid, (b) market-driven selection of Compute Servers for each job, resulting in effective utilization of resources across the grid, and (c) improved utilization within individual Compute Servers.

Utilization of individual Compute Servers is improved by the notions of adaptive jobs *and smarter job schedulers. Server selection is facilitated by quality-of-service (QoS) contracts for parallel jobs. Market efficiencies are then attained by a bidding and evaluation system that makes the Compute Servers compete for every job by submitting bids, thus transforming the computational grid into a free market. Job submission and monitoring is simplified by several tools and databases within the Faucets system.*

We describe the overall architecture of the system. All the essential components of the system have been implemented, which are described in the paper. We also discuss ongoing work and future research issues.

1 Introduction

Over the past decade there has been a dramatic increase in the amount of available computing and storage resources. The emergence of high performance compute clusters has lead to compute power becoming relatively inexpensive and abundant. With high speed networking many geographically distributed resources can be coupled together, thus resulting in the raw infrastructure of the *computational grid.* The presence of powerful parallel machines has fueled the development of large scale and high performance applications. Standardized programming systems such as MPI, collections of libraries such as Linpack, Paramesh[20], Global Array [12], domain specific frameworks[2], and advanced support for irregular and dynamic applications (such as Charm++ [14]) are facilitating development of such applications. The Globus Tool Kit[8] provides middleware to make these applications run on geographically distributed resources.

There are two major hindrances that need to be overcome to fully utilize the *computational grid.* The first problem is that end users of massively parallel applications have the tedious task of discovering available and most suitable resources, uploading files and babysitting their applications (i.e. monitoring progress, restarting from the last checkpoint if the job crashed, and when the machine is about to be taken down, checkpointing the job and moving it to another machine, if possible).

The second problem is that existing supercomputers can remain underutilized because of the nature of parallel applications and the mechanism available for users to submit jobs. The utilization of resources in an environment with multiple users and multiple Compute Servers is affected by external fragmentation as well as internal fragmentation. *External fragmentation* occurs because individual users have access to only a subset of parallel machines. *Internal fragmentation* occurs when a Compute Server schedules existing jobs in such a way that new jobs cannot be scheduled even while many processor resources are idle within the same Compute Server. The following two scenarios illustrate these obstacles.

- *Internal Fragmentation:* Consider a single parallel machine with 1000 processors. A user wants to run an urgent and important job **A** which needs 600 processors. However, the machine happens to be running a relatively unimportant but long job **B** on 500 processors. So the important job languishes while 500 processors remain idle.

- *External Fragmentation:* Now consider the following scenario: when a user needs to run a parallel application, all the parallel machines that they have accounts on are busy running important jobs. However, there are several other parallel machines that are idle, but cannot be used since the user does not have an account on them.

Such wastage is clearly undesirable, especially with parallel systems becoming profit centers that sell compute power.

In this paper we present the *Faucets* system which is aimed at simplifying the process of submitting and monitoring jobs for the end user and eliminating inefficiencies in resource utilization. The term *Faucets* draws an analogy with the distribution of water as a utility delivered via faucets.

We contend that the underutilization problem can be solved by treating compute power as a commodity, and by unleashing a market economy for the producers and consumers of compute power. The producers would be the Compute Servers which would run applications for the users and charge the users for the runs.

In the scenario we envisage, users authenticated by a central service submit jobs to the "grid". A job runs on some anonymous supercomputer, and may be moved between supercomputers during its life. A few rare large jobs may also run on multiple supercomputers simultaneously. Users can monitor and interact with their jobs via the Web, and input and output files can be appropriately moved from and to the user's computer. Users pay for the compute power used via the billing services, or barter the unused compute power of their own Compute Server via an accounting service that allows Compute Servers to pool their resources effectively. The basic architecture of the Faucets system for supporting this scenario is described in Section 2.

We plan to improve resource utilization in this scenario by two interdependent classes of mechanisms:

1. Optimizing the usage of each individual parallel machine with smart job schedulers, and

2. Providing automatic, scalable and market-efficient matching between jobs and Compute Servers.

To this end, we develop the notion of **quality of service (QoS) contracts** for parallel jobs (see Section 2.1), which specify the job's resource requirements, its behavior over the range of processors it can use, as well as its payoff, i.e. how much the client will pay for running the job.

An important twist we add to this scenario is the introduction of **Adaptive Jobs:** which can change the number of processors allocated to them at runtime on demand[15]. In a previous paper [15], we described how the Charm++ load balancing framework can be extended to create adaptive jobs. Traditional MPI jobs can also be transformed into adaptive ones via our adaptive implementation of MPI (AMPI [3]).

Resource utilization on individual parallel machines can now be optimized by **smart job schedulers** that shrink and expand the processors allocated to their jobs as needed. In the example above, job B can be shrunk to 400 processors to make the remaining 600 processors available for the important job A, thus fully utilizing the system.

While operating as profit centers in the market economy of parallel jobs, such smart schedulers can further tune their processor allocation to maximize their profit by taking complex payoff functions into account, which may specify premiums for early completion and penalties for delays beyond deadlines. Further, the schedulers incorporate **bid generation algorithms** (Section 5.2), which take into account the current commitments and the grid job "weather" (analogous to the network weather systems [25], and/or "futures" market for perishable commodities) to generate bids for jobs submitted by users. The faucets system must scalably select machines that potentially match the requirements of the jobs, and scalably evaluate and select bids on behalf of users.

2 Architecture

In the near future the *computational grid* may consist of tens of thousands of Compute Servers each with its distinct way of functioning. Potentially, millions of jobs, each with a QoS requirement, may be submitted to the grid per day. Resource allocation in such a scenario becomes the problem of matching between the jobs with the available Compute Servers in an efficient (both from the point of view of the client as well as the Compute Servers) and scalable manner.

We have developed a prototype *Faucets* system to experiment with the ideas mentioned earlier. Figure 2 shows a simplified overview of the architecture of our Faucets system.

The main components of the system are Central Faucets Server(FS), Faucets Daemon(FD), Adaptive Queueing System aka Scheduler aka Cluster Manager (CM), Faucets Client (FC), AppSpector Server(AS) and Database(DB).

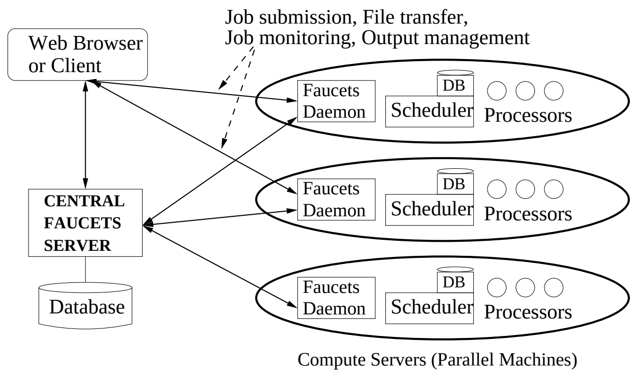

Job submission, File transfer,
Job monitoring, Output management
Web Browser or Client
CENTRAL FAUCETS SERVER
Database
Faucets Daemon
DB
Scheduler
Processors
Compute Servers (Parallel Machines)

Figure 1. System Components.

Each Scheduler (CM) represents an individual Compute Server in the system. The Scheduler running on the supercomputer can be a traditional queuing system, or the Adaptive Job Scheduler in Section 4. Each scheduler is associated with a Database(DB) to store the current status of all the running and scheduled jobs on the Compute Server. This Database will be queried for determining the state of the scheduler, based on which the Scheduler has to decide whether to accept a new job or not. Each Scheduler is associated with a Faucets Daemon process which listens on a well-known port. The FD acts like an agent for the Scheduler to communicate with the rest of the Faucets system. At startup each FD registers itself with the Faucets Central Server(FS). The client process sees the FD, but not the actual CM. When FD receives a *bid request* from a client, it queries the CM with that request and receives an appropriate bid which it forwards to the client. The client then chooses the least bid and submits its job to the corresponding FD which in turn starts the job through the CM. In essence to the external world, FD is the representative of the Compute Server to the faucets system.

The Faucets Central Server (FS) is at the heart of the system. It maintains the list of available Compute Servers and refreshes the list by periodically polling the corresponding FDs. The FS also maintains the list of applications clients can run. In addition the FS is also responsible for authenticating the users of the system. It uses a database to store the Users information. This database also stores the directory of available Compute Servers and some information about each one, such as the maximum number of processors it has, the available memory, CPU type, and the address and port number of the FD corresponding to each Compute Server. When the client contacts FS for the list of available Compute Servers, FS returns this directory of FDs.

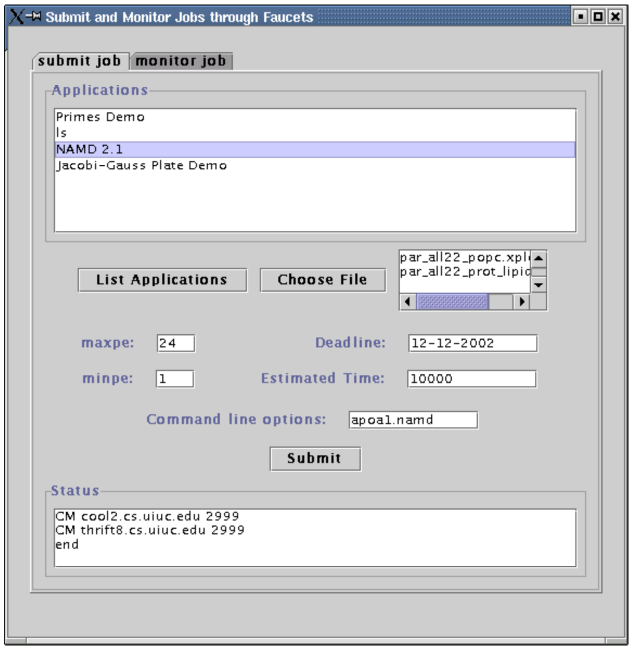
The user interacts with the system using a web browser or a command-line client or a GUI client by authenticating himself to the Faucets Central Server. To submit a job, the client connects to the Central Server and requests a list of matching supercomputers. The client then connects to each FD and solicits a bid for the desired job. After some interaction between the FD and the Scheduler, the FD either declines the job or replies with a bid. Once the bids are collected, the client chooses a satisfactory bid, and informs the appropriate FD. At this point the client uploads the input files to the chosen FD and the FD takes over the job and starts it on the Scheduler. Once the job starts, the FD registers the running job with the AppSpector Server(AS). Figure 2 shows our GUI client through which the user specifies the application name, parameters such as minimum (minpe) and maximum (maxpe) number of processors, estimated time, deadline and files to be uploaded to the parallel machine.

AppSpector is the *Job Monitoring* component of the Faucets system. AppSpector server connects to the job through a network connection and buffers the display data so that multiple clients can monitor the job simultaneously. Any authenticated users using the faucets client can connect to their running (or just completed) parallel job using its job-ID via the AppSpector. The AppSpector retrieves dynamic output from the parallel program and provides a graphical representation of the current status to the user, as shown in Figure 3. One section of this display is application specific and the other section generic, providing the processor utilization/throughput of the application on the Compute Server. At any point of the job execution the user can download the output files generated by the job.

2.1 Specifying QOS Requirements

The Job requirements portion of the quality-of-service contract must include, at a bare minimum:

- The software environment required by the job. This could include the executable for the job, the host operating system, and the required compilers and libraries.
- The number of processors the job can run on (a single number, a set of numbers, or a range.)
- The amount of time needed to complete the job, and some notion of how this changes with the number of processors.
- The job's payoff; and how this changes with the completion time of the job. Thus a job with a


Submit and Monitor Jobs through Faucets

submit job | monitor job

Applications

Primes Demo
ls
NAMD 2.1
Jacobi-Gauss Plate Demo

List Applications | Choose File

par_all22_popc.xpl
par_all22_prot_lipi

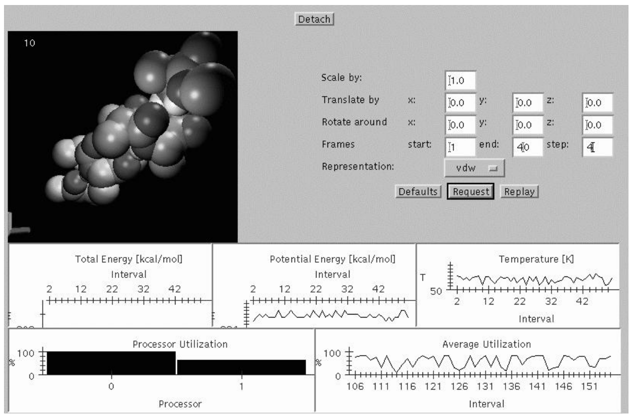
maxpe: 24
Deadline: 12-12-2002
minpe: 1
Estimated Time: 10000
Command line options: apoa1.namd

Submit

Status

```
CM cool2.cs.uiuc.edu 2999
CM thrift8.cs.uiuc.edu 2999
end
```

Figure 2. Faucets Client: Job Submission



Scale by: 1.0
Translate by x: 0.0 y: 0.0 z: 0.0
Rotate around x: 0.0 y: 0.0 z: 0.0
Frames start: 1 end: 40 step: 4
Representation: vdw

Figure 3. Faucets Client:AppSpector Display of a NAMD Job

deadline would have a steep post-deadline drop-off in the payoff vs. time function.

In the current implementation, the QoS requirements are simplified to include only the following: the minimum and maximum number of processors, per-processor and total memory requirement, total CPU time, or wall-clock total time, (optionally) the efficiency with minimum and maximum number of prcessors (with linear interpolation assumed), and a deadline. In addition, an experimental feature includes a payoff functions with a soft and hard deadline with specification of relative payoff as a function of time of completion. (payoff at soft deadline, payoff at hard deadline, and penalty after deadline. Linear interpolation is assumed for payoff between soft and hard deadlines).

One of the research issues is to decide the level of detail in the specifications. For example, the completion time as a function of number of processors can be specified by simple linear function (as used above) or more sophisticated models.

Job requirements should also be made machine-independent. So in a scientific application, for example, one might specify the run time as the floating-point operation count times the machine speed divided by the parallel efficiency. An exact answer is, of course, nearly impossible to obtain; but easy to estimate bounds on these quantities are all that are needed.

Other requirements include memory, communication and disk access patterns. For memory, one could list the amount of memory used per processor, as well some measure of the cache utilization. For communication, one may characterize the number and volume of messages internal to the program, and the communication with outside world, either for continuous interaction or initial/final data transfer. Disk I/O may also be characterized by initial input size, output size, and access pattern during the run itself.[1]

Some applications have distinct phases or components, each with very different requirements. They can potentially be housed on different supercomputers over time. Even when they are running on the same machine, the scheduler may benefit from knowing the shift in performance parameters when the program shifts from one phase to another. The QoS contract will be able to specify such phases and components, and iterative structures around them (if any). Note that to be useful, such a phase must last for several minutes, to justify the overhead of moving the job.

[1]For example, an out-of-core solver may have extensive online disk usage.

2.2 Security Model and Other Considerations

The client authenticates itself to the Faucets Server through a userid, password pair. So every user should obtain an account from the Faucets system. Also the client embeds the userid,password information in any communication with the FS or any FD. But since the FD does not have any accounting information, it contacts the Faucets Central Server again to verify the user's authenticity. In future,the single sign-on feature of the Globus Security Framework(GSI) [10] can be used to avoid these multiple authentication requests to the Faucets Server for the same user.

In many scenarios envisaged in Faucets, the end-user may not have an account with the particular Compute Server that is to run their job. The Faucets system runs the job with a temporary userid. This makes the issue of Compute Server's trust in the clients job more complicated. However, we resolve this issue by identifying classes of secure applications, and allowing each cluster to adopt a different policy and export a different set of applications.

One proposed mechanism is *Mobile Sandboxing*, as described in [17], to trap such untrusted calls and deny service. Also, since each site may have different policies of trust, these Sandboxes have to be configured differently for each Compute Server.

The emergence of JAVA into the realm of high performance computing provides us another alternative class of secure applications.

Further, many individual Compute Servers may have their own applications that the administrators trust based on their knowledge of those applications (e.g. locally developed applications). They may include such applications in their list of supported applications. The Faucets Server can then keep track of registered applications from each Compute Server to avoid unnecessary broadcasts of request-for-bids.

In our current prototype, we assume that each parallel machine exports certain "Known Applications" and the user will be able to select one of these available jobs through the Faucets Server. This scheme can be generalized to a auditing-and-certification service for applications. The recent developments and standardization of *WebServices* provides us a better way of doing this. On the lines of the Bayanihan Computing .NET [7], each Compute Server can potentially provide a WebService which allows the *Computational Clients* to choose an application, execute them and retrieve the results. Open Grid Services Architecture (OGSA)[26] describes a standard way of integrating the Grid and Web Technologies. GSI can enable web browsers to single sign-on to multiple Web Servers and also to delegate capabilities to a web server so that the server could act on the client's behalf. Faucets system can significantly leverage the new OGSA enabled Globus Tool Kit components for integration of our system with WebServices.

3 Improving Ease of Use

The Faucet systems also aims at simplifying and automating the process of running parallel applications for the end user. For one, users don't need to learn the peculiarities of multiple queuing systems (and their installations) used by different Compute Servers. Input files are uploaded and outputs downloaded automatically. As discussed in the section on Architecture, the web-based Appspector subsystem makes it easy to interact with parallel jobs, irrespective of where they are running. We are also implementing features that allow the system to restart users jobs from their last checkpoint if the system had to stop the job or if the machine had any transient hardware problem. We will also make use of Data Grid [28] components of Globus for managing files, for the purpose of storage and visualizations.

4 Improving the Utility of a Server

Individual Compute Servers must be able to maximize their utilization (and profit, in the general case). The job schedulers in Faucets system make use of *adaptive jobs* to this end.

An **adaptive job** is a parallel program that can dynamically (i.e. at run-time) shrink or expand the number of processors it is running on, in response to an external command or an internal event. The number of processors can vary within the bounds specified when the job is started. Typically, the user will specify the bounds taking into consideration memory usage and efficiency of the job on a given number of processors. We have developed adaptive jobs in both Charm++ and MPI. The performance and implementation of adaptive jobs is presented in [15].

4.1 Smart Job Scheduling

Once adaptive jobs are feasible, with their characteristics and payoff functions specified in the QoS, they enable the design of intelligent job schedulers that aim at maximizing system utility. Most current production queuing systems are incapable of exploiting the opportunities created by adaptive jobs. We have developed an adaptive job scheduler that can manage such adaptive jobs [15].

The scheduler is triggered when a new job arrives in the system, and when a running job finishes (or requests a change in the number of processors assigned to it). On arrival, the scheduler analyzes the job's resource requirements and deadlines to decide if it can be accepted.

The scheduler would try to maximize a system utility metric. This metric can be System Utilization, Job response time, or a more complex profit metric, specifying the amount the user pays if his job is finished before the deadline. Hence if a high profit job arrives and has a tight deadline the low priority jobs can be shrunk and the freed processors can be allocated to the high priority job. The communication topology also needs to be considered because the shrunk jobs should remain to have locality and a contiguous set of processors need to be assigned to the new job. Jobs may also have to be check-pointed and restarted at a later point in time and possibly at another (subcontracted) Compute Server with a different architecture.

Decisions on allocating processors to jobs is taken by a strategy that can be plugged in to the adaptive job scheduler. One of the earliest strategy we implemented is presented in [15]. It is a simple strategy that tries to maximize system utilization, by using a variant of equipartitioning: Each job gets a proportionate shared of available processors, while respecting the specified upper and lower bounds on the number of processors for each job.

The utility metric can also be maximizing the payoff function from running a job before its deadline: Such jobs typically have a soft deadline, and a hard deadline. The payoff for the job linearly decreases after the soft deadline, and may have a significant penalty after the hard deadline. In such a scenario, running a new job may delay other jobs and lead to a loss in profit. So the payoff from the new job must at least compensate for the loss mentioned above or the job must be rejected. The strategy must find time windows for the job in its processor-time Gantt chart before the job's deadline. If enough time cannot be allocated for the job it must be rejected. We are currently in the process of developing such a strategy. Our current prototype strategy accepts a job if it is profitable and can be scheduled to run now or at a finite lookahead in future.

5 Market-Efficient Server Selection

This section describes the components of the Faucets system that decide which job runs on which Compute Server. For each component, we describe its current implementation, and ongoing work. We then also comment on research issues that will arise in scalable implementations in future.

5.1 Scalable Identification of Potential Servers

In the current implementation, the client software (but not the end users personally) gets a list of all Compute Servers from the Central Faucets Server (FS) , and broadcasts a request for bids, with QoS requirements, to all of them. Ongoing work involves implementation of simple filtering services at the FS so that static properties (such as number of processors and amount of memory per processor) as well as dynamic properties (e.g. current availability of the Compute Server) are taken into account to eliminate Compute Servers from the broadcast. In future, the broadcast itself will be handled by a distributed Faucets system, making the potential-server selection scale up, even in the presence of millions of jobs submissions a day.

5.2 Bid Generation Algorithms

Probably the most research intensive task will be development of algorithms for generating bids for jobs submitted via the request-for-bids broadcasts. These algorithms will run at individual Compute Servers, and will reflect the characteristics of the Compute Server, its orientation to risk and profit. The bidding decisions can be potentially based on local factors, such as how busy the Compute Server is during the time-period covered by the job, and how far into the future is the job's deadline. For example, a simple strategy may be to set a low bid if the job's deadline is in the very near future (e.g. next hour), and the machine is relatively free.

The current strategies we have implemented include a baseline strategy that always returns a multiplier of "1.0" if it can run the job. (The bid is converted to Dollar amount my multiplying the CPU-seconds needed for the job with a normalized cost and the multiplier returned by the bidding algorithm). Another implemented strategy returns a multiplier linearly interpolated between $k(1-\alpha)$ and $k(1+\beta)$ depending on what the average system utilization is likely to be between the current time and the deadline of the proposed job. k, α and β are parameters of this strategy (current values we use are 1, 0.5 and 2.0). We expect α and β to be associated with the risk the Compute Server is willing to take to maximize profit, and k with urgency of the job for the cluster. (For the job in the example above, with a near-by deadline, one expects to use a low value of k).

In future versions, the bid may also depend on non-local factors, such as "what is the average price of similar contracts in the recent past, in the whole system?" or "how busy is the entire computational grid likely to be during the period covered by the deadline?". For this, bid generators need support from the Faucets system.

5.2.1 Faucets Support for bidding

The Faucets system will provide such global information to Compute Servers, and/or their agents running on faucets infrastructure. The particular mechanisms we envisage supporting include: a history of every individual contract over recent time periods, summaries based on various histogram metrics: (For example, grouping jobs based on the minimum or maximum number of processors they need), trends for future usage based on customer surveys, etc.

5.3 Scalable Bid Evaluation and QoS Contract

In the current implementation, each client receives all the bids, and selects one of the Compute Servers for the job based on a simple criteria (such as least cost, or earliest promised completion time). We expect this scheme to scale to reasonably large grids (consisting of hundreds of Compute Servers). However, in a larger grid of the future, a scalable mechanism is needed for Compute Server selection for a couple of reasons. Firstly, the large number of Compute Servers will make it impractical for each client to deal with a flood of bids. Secondly, since many bid-requests may be in progress at the same time, a two phase protocol will be needed to get a firm commitment from the selected Compute Server (which may have received a more lucrative job in between).

An issue for future research here is a scalable asynchronous system for bid evaluation and contract confirmation. We envisage a system in which each Compute Server as well as client is represented by several agent processes running on the distributed faucets framework. The server agents communicate with their master Faucet Daemons, as well as with bid commitment algorithms of the Faucets system. The client agents simply specify user-specific selection criteria to evaluation.

We plan to publish a generic interface for the bid-generation algorithm, allowing other researchers to test their bid generation algorithms against each other.

5.4 Simulation System

To evaluate the scalability of the framework, and to compare the effectiveness of alternative bidding strategies, we have built a simulation framework: Each entity in the Faucets system — Clients, Compute Servers, Faucets-Server (and its distributed servers in future), job schedulers with their bid-generation algorithms, and application programs — is represented by an object, and discrete-event simulation is carried out over patterns of job submissions under study.

5.5 Alternative Contexts for use of the Faucets

The Faucets system can be used in a variety of possibly overlapping contexts, as described below.

5.5.1 Pay-for-use system

Clearly, the primary aim of the Faucets system will be in a context in which users will pay for running each job. The bids in this context will be Dollar amounts. It may be necessary to have regulatory mechanisms in place to avoid misuse of markets: limits on how far the bids can be from some notion of "normal" price can be one such mechanism. It may also be necessary to have additional priority to jobs of national importance to prevent denial-of-service attacks on such systems.

5.5.2 Academic Applications

How can the current practices in the Science and Engineering academic research transition to a market based economy? This research suggests that management of cycle-providing centers be decoupled from users and handed over to private (for-profit) producers. Users can then be allocated quota in terms of Service-Units (SUs) as before. However, the bids generated by the Compute Servers will now be multipliers to SUs rather than Dollar amounts. ("I will run your job that needs 1000 SUs, but I will charge 1400 SUs for it.", or "I will only charge 750 SUs for it"). This will lead to bringing market efficiency to such centers. Traditional supercomputing centers supported by NSF can still be in the business of providing expert services and consulting for effective parallelization of parallel jobs.

5.5.3 Bartering

The Faucets architecture is equally suitable to create cooperative computing environments where a community of individuals share each others' resources. Those who are contributing to a common pool can get access to that pool [23]. We have a reasonably sophisticated accounting system in place for deciding how much resources each contributor can get. Each contributor earns *credit* for sharing his/her resource and can use up the credit when needed. In our architecture the *Faucets Central Server* keeps track of the credits of all the collaborating clusters. Each user belongs to a single *Home Cluster* and normally whenever he tries to submit a job, the system tries to submit the job to the user's Home Cluster. But if the resources on the Home Cluster are not available and the Home Cluster has enough credits the system tries to submit the job to any of the collaborating Compute Servers and the appropriate number of credits are added to the Compute Server that executed the job and equal amount is deducted from the Home Cluster's account. The credits can be amount of the computational units the job has taken to execute or any other function of it. This mode of bartering of computational units is very useful when the participating entities in the Grid have to be both service provides and consumers.

5.5.4 Intranets

When a company or a laboratory wishes its Compute Server's resources to be pooled among its users, the Faucets system can be used with some small modifications. Different jobs may have priorities assigned by management. Pre-emption of low priority jobs may be allowed (with automatic restart from a checkpoint later). Further, some elements of the bartering scheme may be incorporated in order to allow individual departments or users from getting "fair usage" from resources, so that high priority jobs do not forever starve a subset of users, who may own some of the resources,

6 Related Work

Several projects have studied resource allocation and job management on the Computational Grid. Condor-G [11], Legion [1], SETI@home, Entropia's PC Grid Computing, Parabon's Pioneer, NetSolve [9], Application Level Scheduling (AppLeS)[6], and Bayanihan [16] are distributed and ubiquitous resource allocation frameworks.

Condor-G [11] gives end-users a unified view of all the dispersed resources they are authorized to use. It gathers information about the Grid resources and the job requests from the users in the form of ClassAds and with the help of the Matchmaking framework [24] it tries to determine where to execute user jobs. The ClassAds are similar to the QoS contracts and the Bids generated by the Faucets system. With ClassAds, the system matches jobs to resources; In contrast, with Faucets, choosing the appropriate "bid" (and hence the Compute Server) is up to the client (or Client's Agent). The Condor-G match making algorithms are applicable in our context at the stage of screening the Compute Servers to get a list of potential Compute Servers.

Legion [1] takes an object oriented approach to the problem of resource scheduling, by formulating an object placement process model. The Coordinator/Mapper (CM) is responsible for making the mapping decision. Legion concentrates more on providing basic infrastructure for the grid similar to Globus[8]. SETI@home, Entropia's PC Grid Computing, Parabon's Pioneer and Bayanihan [16, 7] employ a master-slave paradigm where a single master server controls the distribution of available work among different worker agents sitting on the users desktops. NetSolve [9] is a client-agent-server RPC-based system to solve computational problems. Its middleware system provides a computational framework for "task farming" applications with load balancing and scheduling strategies to distribute tasks evenly. NetSolve scheduling strategies do not have the economy model built into them. Both NetSolve and Nimrod concentrate on scheduling master-slave based loosely coupled independent tasks. Faucets, on the other hand, focuses on parallel jobs. AppLeS (Application Level Scheduling)[6] on the other hand takes the approach of developing individual scheduling agents for each application, thus adapting them to the execution-time characteristics of dynamic, distributed environments. The user must provide an application-specific performance model, describing its structure and execution activities, and the application performance criteria.

Many resource allocation frameworks have also studied and incorporated *computational economics and computational markets* [18, 19, 22, 4, 27, 5, 13, 21]. Enterprise [27] is one of the earliest decentralized market-like schedulers for load sharing in distributed computational environments. The protocol has *announcement, bid, and award* stages. In the *announcement* stage, a *client* broadcasts a request for bids which includes a description of the task to be run, an estimate of required processing time, and a numeric task priority. Idle *contractors* reply with bids containing estimated completion times for the client's announced task. The client collects bids from responding contractors, evaluates all the bids, and awards its task to the best bidder (with the earliest estimated completion time). Since Enterprise has no concept of market price, the flexibility of the system is inhibited by constraining the criteria by which contractors and clients could make decisions. Also, it is limited to the execution of independent tasks on compatible workstations. Spawn [5] takes a slightly different approach, where a seller executes an *auction* (sealed, second-price auctions) process to manage the sale of his workstation's processing resources, and a buyer executes an application that *bids* for time on nearby auctions. Nimrod/G [22] has a computational economy based distributed scheduling component that tries to select the resources that meet the deadline and minimize the cost of computation. In addition to addressing these issues, *Faucets* system aids resource providers in their price decision process (and hence scheduling), to maximize their profit and to attract more customers.

7 Summary and Future Work

We described *Faucets*, a system that supports the notion of compute power as a utility. Faucets is designed for parallel jobs, although it can also be used for sequential jobs. Users submit jobs with their Quality-of-Service requirements to the Faucets system, via clients (command-line, GUI, or Browser based clients are supported and provided). The Faucets system identifies the Compute Servers that may be able to run the job, and sends the QoS requirements to them. The Compute Servers themselves run a daemon that interfaces with the Faucets system, and mediates with the local scheduler. The local scheduler may submit a bid for the job. The Faucets system, taking the users-specified selection criteria into account, selects the best bid. Users job files are then uploaded to the system. Although the user doesn't explicitly know which Compute Server is running their job, they can connect to their job, and examine its output and current status via a web-based server called AppSpector. The output files can be downloaded to the user computer (or other storage on the network).

One unique aspect of the Faucets system is its use of adaptive jobs, which can change the number of processors they use on command. Adaptive Queuing Systems (Schedulers) that take advantage of such jobs have been designed for use within the Compute Servers of the Faucets system. Such Schedulers have a competitive advantage over normal schedulers in the free-market economy of compute power unleashed by the grid via Faucets.

The Faucets system is in operation at the University of Illinois, with two research clusters. Soon, it will be extended to several other clusters, some via a bartering subsystem of Faucets.

We described architecture of the Faucets system, and its components. We also identified research issues for future, which involve more sophisticated bid-generation algorithms and scalable bid-evaluation framework. Further, improvements that simplify parallel job administration for the user are being implemented in the system. As a result, we expect Faucets to be a widely used system for utilizing resources on the computational grid.

References

[1] A.S.Grimsaw and W.A.Wulf. Legion: A view from 50,000 feet. In *Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[2] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970, pages 385–395. Springer Verlag, December 2000.

[3] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[4] B.N.Chun and D.E.Culler. REXEC: A decentralized, secure remote execution environment for clusters. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 1–14, 2000.

[5] C.A.Waldspurger, T.Hogg, B.A.Huberman, J.O.Kephart, and W.S.Stornetta. Spawn: A distributed computational economy. *Software Engineering*, 18(2):103–117, 1992.

[6] D.Zagorodnov, F.Berman, and R.Wolski. Application scheduling on the information power Grid. Technical Report CS2000-0644, 11, 2000.

[7] L.F.G.Sarmenta et.al. Bayanihan computing .NET: Grid computing with XML web services. In *Global and Peer-to-Peer Computing Workshop at CCGrid*, 2002.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] H.Casanova, M.Kim, J.S.Plank, and J.J.Dongarra. Adaptive scheduling for task farming with grid middleware. *The International Journal of High Performance Computing Applications*, 13(3):231–240, Fall 1999.

[10] I.T.Foster, C.Kesselman, G.Tsudik, and S.Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[11] J.Frey, T.Tannenbaum, I.Foster, M.Livny, and S.Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.

[12] J.Nielocha, R.Harrison, and R.Littlefield. Global arrays: A portable shared-memory programming model for distributed memory computers. In *Supercomputing*, 1994.

[13] J.S.Chase, D.C.Anderson, P.N.Thakar, A.Vahdat, and R.P.Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[14] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[15] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. A malleable-job system for timeshared parallel machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.

[16] L.F.G.Sarmenta. Bayanihan: Web-based volunteer computing using java. In *Proc. of the 2nd International Conference on World-Wide Computing and its Applications*, 1998.

[17] M.Litzkow, T.Tannenbaum, J.Basney, and M.Livny. Checkpoint and migration of unix processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison Computer Sciences, April 1997.

[18] M.S.Miller and K.E.Drexler. Markets and computation: Agoric open systems. *The Ecology of Computation Ed. B. A. Huberman. North-Holland*, pages 133–176, 1988.

[19] M.Stonebraker, R.Devine, M.Kornacker, W.Litwin, A.Pfeffer, A.Sah, and C.Staelin. An economic paradigm for query processing and data migration in Mariposa. In *Third International Conference on Parallel and Distributed Information Systems, Austin, TX*, 1994.

[20] P.MacNeice, K.M.Olson, C.Mobarry, R.deFainchtein, and C.Packer. Paramesh : A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126:330–354, 2000.

[21] R.Braynard, D.Kostic, A.Rodriguez, J.Chase, and A.Vahdat. Opus: An overlay peer utility service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, 2002.

[22] R.Buyya, D.Abramson, and J.Giddy. Nimrod/G: An architecture of a resource management and scheduling system in a global computational grid. In *HPC Asia 2000*, pages 283–289, 2000.

[23] R.Buyya, D.Abramson, J.Giddy, and H.Stockinger. Economic models for resource management and scheduling in grid computing. *The Journal of Concurrency and Computation: Practice and Experience (CCPE), Wiley Press*, 2002.

[24] R.Raman, M.Livny, and M.H.Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, 1998.

[25] R.Wolski, N.T.Spring, and J.Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[26] S.Tuecke, K.Czajkowski, I.Foster, J.Rey, F.Steve, and G.Carl. Grid service specification, 2002.

[27] T.Malone et al. Enterprise: A market-like task scheduller for distributed computing environments. *In The Ecology of Computation, B.A. Huberman Ed., 40 Amsterdam, North-Holland*, pages 177–205, 1988.

[28] W.Allcock, J.Bester, J.Bresnahan, A.Chervenak, I.Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. Data management and transfer in highperformance computational grid environments. *Parallel Computing*, 2001.