Dispelling Seven Myths about Grid Resource Management

Anand Natrajan, Andrew S. Grimshaw, Marty A. Humphrey, Anh Nguyen-Tuong
Department of Computer Science, University of Virginia
{anand, grimshaw, humphrey, nguyen}@cs.virginia.edu

Grid resource management is often viewed as scheduling for large, long-running, compute-intensive, parallel applications. This view is narrow, as we will argue in this article. Grids today encompass diverse resources including machines, users and applications. Moreover, grid users make different demands from different resources. Therefore, grid infrastructures must adopt a greater responsibility than before for managing resources. Grid resource management cannot mean just scheduling jobs on the fastest machines, but must also include scheduling special jobs on matched machines, preserving site autonomy, determining usage policies, respecting permissions for use and so on. In this article, we will present seven "myths" or common beliefs about grid resource management, and dispel each myth by presenting counter-examples or "observations". These observations have been culled from our experience as well as work done by several experts in major grid-related projects. In order to relate these observations to concrete implementation, we will also present grid resource management in Legion, a grid infrastructure project initiated at the University of Virginia.

What is grid computing?

Grid computing is the ability to use and control network-connected resources that are heterogeneous, distributed, managed independently and potentially faulty. Grid computing has received a lot of press recently with vendors such as IBM, Sun and HP each putting forth their grid strategies. One criticial aspect of grid computing, particularly for large-scale scientific applications, is resource management.

Grids are collections of interconnected resources harnessed together in order to satisfy various needs of users. The resources may be administered by different organisations and may be widely-distributed, heterogeneous and fault-prone. Previously, grid resource management meant finding CPU cycles for running long, compute-intensive, parallel jobs. However, *grid resource management* now means the manner in which resources are allocated, assigned, authenticated, authorised, assured, accessed, accounted and audited. We will contrast these two views of grid resource management in this article. In §1, we will present the beliefs of the previous view as *myths* because we find no basis for their continuance. We will also dispel these myths with counter-examples. In doing so, we will fashion the current view of grid resource management as a collection of *observations* that encompass the meaning of grid resource management and outline the tasks associated with it. When presenting this view, we will draw on the wealth of experience present in the grid community. Several grid projects share the observations we make here.

Grid resource management is the manner in which resources are allocated, assigned, authenticated, authorised, assured, accessed, accounted and audited.

We implemented the current view of grid resource management in Legion, a software infrastructure developed originally at the University of Virginia in the mid-1990s [GRIM97]. In 2001, a company called AVAKI Corporation was founded to provide commercial grid solutions to companies. The underlying architecture of Avaki 2.x is the same as that of Legion, and our discussion here applies to both. A key feature of Legion is its resource management framework. Grid resource management is a complex task, involving security and fault-tolerance as well as scheduling. Grid scheduling itself requires not a one-size-fits-all scheduler but an architectural framework that can accommodate different schedulers for different classes of problems. We will present the broad architectural features of the Legion resource management framework in §2. In §3, we will discuss some of the lessons we learnt in Legion regarding grid resource management.

1. Myths about Grid Resource Management

The underlying theme in the seven myths that are the topic of this article is that, mistakenly, grid computing is considered synonymous with high-performance computing. This theme arose from the high-performance origins of grids in the early 1990s. At that time, one motivation for creating grids was to share CPU cycles across multiple organisations. The idea was to write an application and have it run on remote resources that presumably could finish the job faster. However, since that original concept, grids have changed. Now, the emphasis is more on sharing, be it sharing CPU cycles in the original high-performance sense or sharing files and databases in the more recent data-grid sense. Even applications are shareable entities; an application written by a grid user in California should be available to a user in Japan, provided the requisite security and licensing considerations are met. Each of the myths below disentangles one of the misconceptions that contributes to the main theme.

For each of the myths we present, we will attempt to answer the following questions:

1. *What is the myth?*

2. *What is the proof that this myth exists?*
3. *Why did this myth arise?*
4. *What are the implications of this myth?*
5. *Why is the myth false?*
6. *What is our observation?*
7. *What are the implications of this observation?*

In §2, we will answer:

8. *How did we implement the observation in Legion?*

Forthwith, here are the seven myths.

Myth 1 *Grid Resources = Machines (or CPUs)*

This myth refers to the misconception that only machines, specifically the CPU cycles on machines, are interesting resources on grids. Given that grids were originally meant for sharing CPU cycles, this myth is unsurprising. Early technical literature in grid resource management typically considers only CPU cycles for scheduling jobs. Traditional scheduling algorithms, such as first-in-first out (FIFO), shortest job first and fair-share all consider CPU cycles as the metric to optimise. The myth that CPU cycles are the only resources of interest arises from the mistaken belief that grids are merely cross-domain extensions of “big iron” systems such as multi-CPU machines or large clusters. This belief directly implies that non-CPU resources are relatively plentiful or irrelevant for scheduling, and a grid infrastructure need be only slightly more sophisticated than a cluster management infrastructure. This view ignores other hardware resources such as network bandwidth, disk, memory, swap space or specialised instruments that may be vital for running jobs on a machine. It also ignores valid resources that may be irrelevant for high-performance computing specifically, but relevant for grid computing in general, such as data, as we will argue again and again in this article. As an off-centre example, consider mobile devices. Currently, mobile devices occupy a second-class status as grid resources; they are considered for little more than web portals or for browsing a grid. We strongly believe that the day is not far when mobile devices will be first-class grid resources, providing anything from short-order CPU cycles to on-site data. As a result, they and other resources must be brought into the ambit of a “grid resource” so that full-fledged resource management can be applied to them as well. A better restatement of the above myth leads to our first observation:

Obs. 1 *Grid Resources = CPUs + Disk + Memory + Files + Databases + Users + Administrators + Applications + Licences + Running Jobs + Mobile Devices + …*

A valid question is “By the definition above, what is *not* a grid resource?” A grid resource is anything that may be controlled (see definition above for grid resource management), and which has an Application Programming Interface (API) for doing so. Conceptually, a toaster oven could be a grid resource if (i) we wish to control who toasts what when and how in it, and (ii) this control can be implemented programmatically. Currently, we may neither wish to exert such control over toaster ovens nor avail of standard APIs for them; therefore, toaster ovens, and many devices of the same ilk, are not grid resources. Can people be resources? Certainly, as echoed in the observation above. Typically, people are represented by credentials and authorisations in a grid. These credentials and authorisations can be controlled programmatically, thus effectively controlling a human's actions on a grid.

Data items are key resources in a grid. A recurring use-case scenario among grid users is the requirement to access data that may be located remotely. The emphasis here is not on running applications fast, but on accessing hard-to-reach data. The more ecumenical interpretation of grid resources in the observation above opens the door for resource management strategies involving data placement, replication, tool location, licence management, job monitoring tools, etc. On the other hand, the over-emphasis on CPU cycles as grid resources leads to the next myth.

Myth 2 *Grid Resource Management = Scheduling on a Grid*

The belief that resource management is the same as scheduling is fallacious on two grounds. First, the notion of scheduling may not make sense for large classes of non-CPU resources (see Obs. 1). For example, we may not "schedule" users or data, but we can "authenticate" users or "replicate" data; certainly, the latter tasks manage resources as well. Therefore, focussing only on scheduling effectively ignores most kinds of resources. Second, even if we consider CPU resources alone, resource management is more than scheduling [GAN99] [LIV99]. Classical scheduling targets a bag of jobs that have to run on resources that are essentially available all the time. In such an environment, scheduling is a process of imposing discipline on the users so that each has a fair chance of getting dedicated resources for her job. Queuing systems such as PBS [BAY99], LSF [ZHOU92] [ZHOU93], LoadLeveler [IBM93], NQS [KING92], DQS, Codine and SGE are one version of disciplined usage. However, the scheduling problem is more complex for grids. Grids are often constructed by bringing together CPUs (and other resources) available at and controlled by different organisations [BER99] [FOS99] [GAN99] [LIV99]. Accordingly, before we determine when an application should run on a machine, we need to determine whether or not it is allowed to run there at all. In turn, we must authenticate the grid user who wishes to run the application, check her permissions to determine whether she is authorised to run anything on that machine, check to see whether she can run the application in question and check whether the application has a licence to run on that machine. Even determining when she can run is a complex question. An organisation contributing CPU (and other) resources to a grid may demarcate certain hours of the day during which the resources are unavailable to remote users in order to satisfy the demands of its local users. Therefore, even if a remote user's job percolates to the top of the list of jobs to be run, it may not be able to run if the time of the day is not suitable. Added to all of these issues, resources may fail at any time. A grid infrastructure must continue to function, and if possible, support the user's job during resource failures. While jobs run on remote machines, their resource consumption must be monitored to control against over-use and perhaps also to audit for later billing [HUM01]. In short:

Obs. 2 *Grid Resource Management = Grid Scheduling + Security + Fault-tolerance + Site Autonomy + Licensing + Auditing*

Resource management is central to grid computing, and it is a hard problem. Not only does it include the security sub-problem, but it also includes the hard sub-problem of fault-tolerance and the complex sub-problem of scheduling [CZAJ01] [FREY02]. All parts of this problem must be addressed within the architecture of a grid infrastructure. Three major models have been proposed for a grid architecture [FOS99] [FOX99] [GAN99]:

- A model based on commodity-solutions such as HTTP for transport, CGI and JavaBeans for business and session logic, CORBA for resource discovery, and JDBC for database connectivity [FOX99]
- A toolkit model, such as Globus, comprising components that implement basic grid services [FOS99]
- An integrated model, such as Legion, which abstracts all of the resources in a grid and presents the entire grid as a single virtual machine [GRIM97] [GRIM99]

Each approach has its benefits and drawbacks; we refer the reader to the cited literature to evaluate each approach. Briefly, a commodity-based model separates concerns and enables selecting best-of-breed solutions for every sub-problem. However, integrating independently-developed solutions can be painstaking. The toolkit model focusses on underlying tools and mechanisms that are of immediate benefit to users bringing grids incrementally to users. The risk with this approach is that until higher-level tools are built, many tasks that should be performed by the infrastructure have to be performed by users themselves. The integrated model provides layered services ranging from low-level transport up to job submission or data access tools, making deployment straightforward. However, the approach requires tight co-design of components and can be slow to adapt to changing standards.

Myth 3 *Grid Scheduling = Reduction in Application Execution Time*

The most persistent myth about grid scheduling itself is that its goal is to run applications as fast as possible. Reducing the time required to run an application is an alluring goal; therefore, grid scheduling toolkits and interfaces such as Nimrod [ABR95], NetSolve [CASA96] and Ninf [NAK99] have been designed to meet such demands of grid users. Certainly, running applications quickly is commendable, but it is not the be-all of scheduling. High throughput is only one of many criteria that may be important to general grid users when running applications [BER96]. Even if we discount security and fault-tolerance momentarily (see Myth 2: Grid Resource Management = Scheduling on a Grid), users may have more on their mind than just low execution times. For example, for some jobs meeting deadlines may be more important than running fast when they eventually run. In grids with cost models for resource consumption, running cheaply may be most important. For applications requiring special tools or licences to run, scheduling on machines that make those tools or licences available earliest is more important than throughput. When users are developing their applications, running immediately is more important than running fast because of the high probability of an error in the application. Users often getting frustrated when they have to wait for hours or days for their jobs to bubble to the top of a queue only to have them die in the first five minutes because of an error. Humans do make errors; a system that metes draconian punishments (such as loss of research time) for venial errors (such as specifying incorrect parameters) can become unpopular quickly. Therefore:

Obs. 3 *Grid Scheduling = Optimisation_Function(Throughput, Latency, Cost, Deadlines, Fairness, …)*

Grids will be used for a diversity of applications. It is reasonable to expect that different applications will make different demands on a scheduler. Even if we assume that high performance indeed is the only criterion that interests grid users, what kinds of applications can we expect to run on a grid?

Myth 4 *Grid Applications = Parallel Applications*

Parallel applications, i.e., applications whose constituent tasks communicate with each other, are often written to solve large problems. Especially complex instances of such applications have been venerated as “grand challenge” applications. Complex molecular modelling and simulation applications such as Amber and CHARMM [BRO83] [MAC98] often are written using some parallel processing toolkit, such as Parallel Virtual Machine (PVM) [GEI98] or Message Passing Interface (MPI) [HEM99] [SNIR98]. Such applications typically start up a large number of tasks which must communicate in order to make progress. Certainly, grid scheduling should accommodate such applications if possible, because these applications could benefit from running on remote resources. Systems such as AppLeS [BER96], Prophet [WEI95] and others [KEN02] [LIU02] address critical aspects of scheduling parallel jobs, such as partitioning, placement and perhaps load-balancing. Much of this work is directly useful to applications intended to run on a grid. However, focussing on such applications alone is restrictive. First, a large number of interesting applications do not actually fall in this category. Several scientific applications are written to be simply sequential. For example, Basic Local Alignment Search Tool (BLAST) [ALT90], a frequently-used bioinformatics application, compares a protein or genomic sequence against a database *sequentially*. Several other applications are written to be run as *pleasingly-parallel* jobs i.e., these applications are run as multiple jobs that do not communicate with one another. For example, protein database searches and wingflow dynamics calculations are purely pleasingly-parallel applications. Second, most grid users are sophisticated enough to realise that grids or no grids, running highly-communicating tasks over wide-area networks results in very poor performance. Accordingly, they either write their applications as barely-communicating tasks (much like pleasingly-parallel jobs) or require their applications to run on clusters or local-area networks [NAT01C]. Grid schedulers may provide only limited benefit to such users.

Significant effort spent on grid scheduling is focussed on a hard problem called co-scheduling, i.e., scheduling a parallel application across distributed resources such that its tasks start at approximately the same time [SMI00]. Co-scheduling is a necessary process for running a large parallel application on a grid because its tasks must start at roughly the same time to ensure progress and, more importantly, not waste CPU cycles. Co-scheduling becomes especially difficult when the tasks are placed across machines controlled by different organisations, perhaps in different time-zones. Added complexity occurs when the machines have different load and fault characteristics. However, the "last straw" for co-scheduling is the problem of starting an application such that different tasks are started by different queuing systems. Co-scheduling involving queuing systems is prohibitively hard; hardly any queuing system can guarantee when a job will really start unless its human controllers reserve specific times or days for the execution of such jobs alone. Co-scheduling is a very hard problem; however we believe the fraction of grid applications that require co-scheduling is small. Most users are aware of the difficulties involved in co-scheduling and that the benefits are often small because the latencies involved in large network distances can nullify most of the gains accrued by scheduling on powerful remote machines. Accordingly, they restructure their applications to require as little communication as possible (perhaps becoming pleasingly-parallel), or to require fewer resources (thus becoming amenable to within-cluster scheduling). Therefore, while co-scheduling research is beneficial for applications that cannot be restructured in this fashion, immediate and greater benefits can be achieved by focussing on simpler applications. Since the set of grid applications includes parallel as well as other applications:

Obs. 4 *Grid Applications* = *Parallel Applications* + *Sequential Applications*
+ *Pleasingly-Parallel Applications* + *Transactions*
+ *Soft Real-Time Applications* + …

The traditional view of a typical grid application as a parallel application needs revision. Next, we will change another related view of a grid application.

Myth 5 *Grid Applications* = *Long-Running Compute-Intensive Jobs*

The romantic view of a grid application is that of a long-running job, perhaps an instance of a "grand challenge" application, carefully scheduled on a resource that will run this application the fastest (see citations for Myth 4: Grid Applications = Parallel Applications). In contrast, increasingly, grid jobs will be short. Already, several applications, especially parameter-space studies, tend to have short individual jobs. For example, in 2000, researchers at Celera Genomics ran roughly 70,000 jobs per day over 600 processors using Platform's LSF queuing system for analysing genome data and performing tasks such as gene discovery [EDW03] [SMI03]. Thus, each job averaged roughly 12 minutes. As a whole, the application ran for a long time, but in terms of the unit of interest for a grid scheduler, each job of the application ran for a relatively short time. Therefore, the criterion of interest for a grid user running such jobs is the ability to execute as many of the jobs as possible at any given time. Typically, this user would not care even if some of his jobs start on relatively slow machines. Rather, he would be more interested in using every available cycle even if it is on a slow but scarcely-used desktop. Sending such jobs to a queuing system is unattractive since the wait and overhead of starting up such short jobs typically overwhelms the duration of the job itself. As grids evolve, we expect them to move into utility computing models. Here, a grid job may be viewed as a transaction lasting a few seconds to a few minutes. Traditional scheduling mechanisms that view jobs as long-running processes will have limited use in such environments. Such mechanisms will become unattractive quickly in the expected régime of short grid jobs, mainly because the duration for scheduling the job may be longer than the duration of the job itself [BER99]. Scheduling mechanisms that find favour will be the ones that recognise diversity in the application mix and use different strategies for different classes of applications. For example, an involved scheduling process for a long job can be amortised over the duration of the job, but a "quick and dirty" process may be sufficient for short jobs.

Obs. 5 *Grid Applications = Long-Running Compute-Intensive Processes*
+ Large Collections of Short Jobs + Frequent Short Transactions + …

Grid management software essentially provides an operating system for distributed systems. A general-purpose operating system typically provides acceptable performance for all classes of jobs instead of performing extremely well for one class of applications but extremely poorly for all other classes. Likewise, a grid infrastructure must provide, through its scheduler or schedulers, acceptable performance (if performance be the criterion of interest to the grid user – see Myth 3: Grid Scheduling = Reduction in Application Execution Time) for all kinds of applications. Focussing on only one class of applications is unduly restrictive (see a similar argument in Myth 4: Grid Applications = Parallel Applications). Focussing on only one kind of scheduling mechanism for that one class of applications is myopic.

Myth 6 *Scheduling Mechanism = Queuing Systems*

Relying on one kind of scheduler alone to run all kinds of jobs is detrimental to users of a grid [LIV99]. Even the most sophisticated queuing system represents only one kind of scheduling decision, namely FIFO; added variations such as priorities, backfill, multi-queues, etc. make the strict FIFO discipline more amenable to a larger class of users. FIFO-based scheduling works in a restricted environment consisting of machines that are cost-equivalent and users who do not care when their jobs start as long as they get dedicated resources when they execute [BER99]. A general grid is not as restrictive. Typically, a grid has heterogeneous resources, i.e., resources of varying capabilities.

Almost certainly, these resources will have different costs associated with them [BUY01]; at the very least, desirability will be a direct function of performance. More likely however, costs such as time available to run on them, mean time to failure, earliest time to start, etc. will be interesting to users. Users may care when their applications start; too late a start for some applications may render the results obsolete. CPU resources today are relatively inexpensive, but appetite for them is large. Opportunistic scheduling strategies such as those employed in Condor [LIT88], SETI@Home and Entropia, that feed that appetite by scavenging for available and idle cycles are already popular and likely to become more so [LIV99]. Therefore:

Obs. 6 *Scheduling Mechanism > Queuing Systems*

Queuing systems are very good at what they do – running jobs one after another on finite resources that are typically cluster-based. However, the needs of all grid users who run jobs cannot be satisfied by queuing systems alone. Queuing systems attempt to optimise throughput at the expense of latency of jobs, and consequently, the patience of users. Queuing systems often ignore deadlines, do not factor in CPU cycle cost, ignore network costs, typically do not account for disk usage and often assume security considerations are addressed by other systems. Most queuing systems also rely on the user to ensure that input data and binaries are staged on the compute machines before the jobs starts. Queuing systems work very well when the bulk of the users in a grid submit long-running, possibly parallel, compute-intensive, fully-debugged, pre-staged jobs in a production environment. No one expects grids to be restricted to such jobs. Then why restrict grid scheduling to queuing?

Myth 7 *Scheduling is a One-Sided Activity*

This myth refers to the perception that scheduling is either a “user-driven” or a “system-driven” process. “User-driven” scheduling refers to a process wherein only actions by the user initiate activity. Typically, single-user systems such as PCs are user-driven. Conversely, “system-driven” scheduling refers to a process wherein an activity is initiated only as long as it meets systemic goals such as utilisation percentage. Typically, cluster system administrators employ system-driven scheduling, e.g., queuing systems, in order to keep CPUs employed as much as possible. In grid environments, scheduling is unlikely to be one-sided. A user sitting at her terminal may not be aware that her CPU cycles are being used by an application initiated by a remote user; therefore user-driven scheduling is not applicable. Likewise, system-driven scheduling is inapplicable because large classes of applications will require to be run in a manner that does not necessarily align with the goals of system administrators. Therefore, grid scheduling strategies should not be confused with local or cluster-based scheduling strategies. A more viable strategy is:

Obs. 7 *Scheduling is a Negotiation between Resource Providers and Consumers*

Scheduling should involve negotiation between resource providers and resource consumers. Resource providers must be able to enforce site autonomy, i.e., they should be able to indicate when they will allow what applications to run and under what conditions. Of course, resource consumers are free to select whatever resources best meet their needs. Two challenges have to be resolved in order to implement resource negotiation. First, we need a resource specification language rich enough to express descriptions of available resources and expressions for procuring them. Second, we need a framework that can incorporate different schedulers that can match resource specifications with resources. Resource specification languages such as ClassAds [RAM98] show promise in describing resources for matchmaking, i.e., the process of matching the requirements of a resource consumer with the characteristics and policies of resource providers.

An important challenge facing grid infrastructure experts is not finding the one scheduler that solves all scheduling problems, but finding a framework that enables incorporating a diversity of schedulers [ARN01] [BER96] [CZAJ01] [KARP96]. The framework must accommodate various scheduling algorithms, each taking into account different grid and application information. A default scheduler that provides decent performance for most of the applications must be present so that novice users can still benefit from the grid. However, experienced users must be able to select the kind and level of scheduling service they desire and must be able to procure the cost of that service.

Myths of Grid Resource Management

1. Grid Resources = Machines (or CPUs)
2. Grid Resource Management = Scheduling on a Grid
3. Grid Scheduling = Reduction in Application Execution Time
4. Grid Applications = Parallel Applications
5. Grid Applications = Long-Running Compute-Intensive Jobs
6. Scheduling Mechanism = Queuing Systems
7. Scheduling is a One-Sided Activity

~

The scheduling problem in grids is sufficiently complex that we do not expect a fully-automated solution to appear shortly. In other words, pessimistic as it may sound, we do not believe that a grid will be able to select the correct scheduling algorithm for an application without any hint, suggestion or help from the user. Therefore, we must enlist a developer's help in running applications, which in turn means that the developer must be exposed to some of the complexity in the decision-making process. Too often, we fear that users must be protected from complex decision-making. Certainly, users must be protected from the drudgery of mundane complexity, such as remembering which password to use at which site, understanding labyrinthine directory structures at each site, staging files and binaries, knowing which tool to use where and so on. However, we feel users may accept beneficial complexity, e.g., complexity in decision-making. We feel that when the benefits of a process become apparent to users, they will participate in resolving complexity inherent in scheduling decisions. Today, the willingness of users to embrace complexity is expended on chores such as file and account management. However, a grid infrastructure that can liberate users from these chores will find users expending energy on making scheduling decisions, which benefit them more profoundly. Virtually every grid user has a bag of tricks to take advantage of existing scheduling mechanisms at local and remote sites. Some of the tricks are:

- running yet-to-be-debugged compute jobs on machines marked for interactive use
- stealing idle cycles on desktops, machines in other time-zones, etc.
- requesting larger-than-necessary time-slices for a job so that a queuing system promotes the small job to a high-performance queue
- requesting more CPUs than necessary for a job, so that a queuing system promotes the small job to a high-performance queue
- requesting fewer CPUs than necessary for a job, so that a queuing system allows the job to be started on an "interactive" queue, and later starting multiple processes on each CPU
- "shotgunning" the application, i.e., requesting it to be run on every machine, waiting for one instance to complete and cancelling the others
- forking off jobs at the back-end so that even if the main process is terminated by the scheduler, the child processes continue to run

Several of these tricks are clever and complex; it is naïve to believe that users who devise such tricks are unwilling to embrace complexity. A grid infrastructure that meets the expectations of users may find them willing to redirect their attention to choosing proper schedulers instead of trying to beat the system.

2. The Legion Approach

The basic architecture of Legion is reflective, object-based and extensible, and is in essence, an operating system for grids. A common, accepted, underlying architecture and a set of necessary services built over it is critical for success in grids.

2.1. History

The Legion Project began in late 1993 with dramatic increases in wide-area network bandwidth looming on the horizon. Other changes such as faster processors, more available memory, more disk space, etc. were expected to follow in the usual way as predicted by Moore's Law. Given the expected changes in the physical infrastructure, we asked what sorts of applications would people want, and what system software infrastructure would be needed to support those applications. The Legion project was born with the determination to build, test, deploy and ultimately transfer to industry, a robust, scalable, grid computing software infrastructure. We worked on a prototype system from 1994 through 1996, obtained first funding in early 1996, and wrote the first line of Legion code in June 1996.

The basic architecture is driven by a core set of principles and requirements we have presented often [GRIM94] [GRIM97] [GRIM98B] [GAN99] [NAT01B] [LEW03]. The resulting architecture is reflective, object-based to facilitate encapsulation, extensible, and is in essence an operating system for grids. We feel strongly that having a common, accepted, underlying architecture and a set of necessary services built over it is critical for success in grids. In this sense, the Legion architecture anticipates the drive to web services and Open Grid Systems Infrastructure (OGSI).

We deployed Legion at the University of Virginia, the San Diego Supercomputing Centre, the National Centre for Supercomputing Applications and the University of California at Berkeley for our first large-scale test and demonstration at SuperComputing in November 1997. In those days, keeping a grid running for a day was challenging. Embarrassing crashes and unusual failures drove home the lesson that the world is not “fail-safe”. In a year’s time, i.e., by SuperComputing in November 1998, we could keep grids up for a month to three months easily. We demonstrated Legion on NPACI-net (NPACI: National Partnership for Advanced Computing Infrastructure), consisting of machines at the University of Virginia, the California Institute of Technology, the University of California at Berkeley, Indiana University, the University of Michigan, the Georgia Institute of Technology, the Tokyo Institute of Technology and the Vrije Universiteit at Amsterdam. By this time, we had ported or helped to port* dozens of applications from diverse areas: materials science, molecular modelling, sequence comparison, ocean modelling and astronomy. NPACI-net continues today with additional sites such as the University of Minnesota, the State University of New York at Binghamton, the Pittsburgh Supercomputing Centre and the San Diego Supercomputing Centre. Supported platforms include Windows 2000, the Cray T3E and T90, AIX, Solaris, Irix, HPUX, Linux, Tru64 and others.

From the beginning of the project, we envisioned a technology transfer phase in which the technology would be transferred from academia to industry. We felt strongly that grid software would move into mainstream business computing only with commercially-supported software, help lines, customer support, services and deployment teams. In 1999, Applied MetaComputing was founded to carry out the technology transition. In 2001, the company raised $20M in venture capital and changed its name to AVAKI. The company acquired legal rights to Legion from the University of Virginia and renamed Legion to Avaki. Avaki was released commercially in September 2001. Avaki is a version of Legion that is hardened, trimmed down and focussed on commercial requirements. Despite the change in name, the core architecture and the operating principles remain the same.

* “Porting” an application to a Legion grid is a simple task; Legion supports running legacy applications on a grid unchanged. Legion does not require re-writing or even re-compiling applications to run on a grid.

2.2. Technology

Legion is an architecture for building a grid infrastructure. This infrastructure consists of cooperating services that present the users with the illusion of a single virtual machine [GRIM97]. Legion has been called "an operating system for grids", and rightly so. Within an integrated environment, as opposed to a commodity-based [FOX99] or toolkit [FOS99] approach, Legion provides standard operating system services – process creation and control, interprocess communication, persistent storage, security and resource management – on a grid. By doing so, Legion abstracts the heterogeneity inherent in distributed resources and makes them look like part of one virtual machine [GRIM98B]. In order to achieve this goal, Legion manages complexity in a number of dimensions. For example, it masks the complexity involved in running on machines with different operating systems and architectures, managed by different software systems, owned by different organisations and located at multiple sites. In addition, Legion provides a user with high-level services in the form of tools for specifying what an application requires and accessing available resources.

As part of the virtual machine abstraction, Legion provides mechanisms to couple diverse applications and resources, vastly simplifying the task of running applications on heterogeneous distributed systems. The virtual machine provides secure shared objects and shared namespaces. Each system and application component in Legion is a named object. The object-based architecture enables modularity, data and fault encapsulation, and replaceability – the ability to change implementations of any component. Legion supports PVM, MPI, C, Fortran (with an object-based parallel dialect) [FERR98], a parallel C++ [GRIM96], Java and the CORBA IDL [SEIG96]. Also, Legion addresses critical issues such as flexibility and extensibility [GRIM98A], site autonomy, binary management and limited forms of fault detection/recovery. From inception, Legion was designed to manage millions of hosts and billions of objects – scales to be expected in a grid [GRIM99].

Security, a key component of grid resource management, was integrated into Legion from the design through implementation [FERR99]. Every Legion object, whether it be a machine, a user, a file, an application or a running job, has a security mechanism associated with it. The mechanisms provided by Legion are general enough to accommodate different kinds of security policies within a single grid. Typically, the security provided is in the form of access control lists. An access control list indicates which objects can call which methods of an object. This fine-grained control mechanism enables users and grid administrators to set sophisticated policies for different objects. The authentication mechanism currently employed by Legion is a public key infrastructure based on key pairs. The keys are used to sign certificates as well as encrypt and decrypt messages securely.

Fault-tolerance, another important resource management component, is implemented in a number of ways in Legion [NGU00]. Basic Legion objects are fault-tolerant because they can be revived consistently and safely after deactivation. When a Legion object is deactivated, it saves its state to persistent storage and frees memory and process state. Subsequently, it may be reactivated from its persistent state. If it is reactivated on a different machine, Legion transfers its state to the new machine whenever possible. In addition, some objects can be replicated for performance or availability. Legion's MPI implementation provides mechanisms for checkpointing, stopping and restarting individual tasks. Finally, Legion provides tools for retrieving intermediate files generated by legacy applications. Users can restart their instances using these intermediate files. Individual jobs of a parameter-space study are monitored and restarted if they fail [NAT02].

Legion is both an infrastructure for grids as well a collection of integrated tools constructed on top of this infrastructure. The basic infrastructure enables secure, dataflow-based, fault-tolerant communication between objects. Communicating objects could be diverse resources, such as applications, jobs, files, directories, schedulers, managers, authentication objects (representations of users in a grid), databases, tools, etc. A typical chain of events in a grid could involve a user initiating a tool to start an instance of an application on a machine. This chain results in a tool object contacting an application object (to create an instance of this application), which in turn contacts a scheduler (to generate schedules for running this application on a machine), which contacts a collection (to procure information about machines), an enactor (to reserve time on the target machine) and a host object (to start the job on the machine) [CHAP99] [KARP96]. After the scheduler selects a host object, it contacts the application object with enough information to start a job instance on the machine. We depict this sequence of events with the timeline in Figure 1.

The figure intentionally abstracts a few important design decisions. *First*, the figure shows only the high-level interaction of objects that participate in a sequence of events related to scheduling. In other words, it does not show the security and fault-tolerance features that are part of every event in that and any chain of events in Legion. The Legion grid resource management framework is not restricted to scheduling alone. Every communication between any pair of objects must go through the Legion protocol stack, which involves constructing program graphs, making method invocations, checking authorisation, assembling or disassembling messages, encrypting messages, retransmitting messages, and so on. We show an example stack in Figure 2. Since every communication goes through such a stack, Legion provides security and fault-tolerance as well as scheduling as part of an integrated resource management framework.

Second, scheduling in Legion is a process of negotiation. Most schedulers view CPU cycles as passive resources waiting to be utilised by the next available job. However, in a multi-organisational framework, a CPU is not necessarily available simply because it is idle. The owner of the CPU – the organisation that controls the machine – may impose restrictions on its usage. Therefore, when matching a job to an available CPU, Legion initiates a negotiation protocol which respects the requirements of the job as well as the restrictions imposed by the CPU owner. In other words, we consider site autonomy an important part of the scheduling, or more correctly, the resource management process. In Figure 1, even if a scheduler selects a particular host for running a job, the host may reject the job based on its current policies. Depending on the implementation, the scheduler may investigate variant schedules or may inform the user of the failure to run the job.

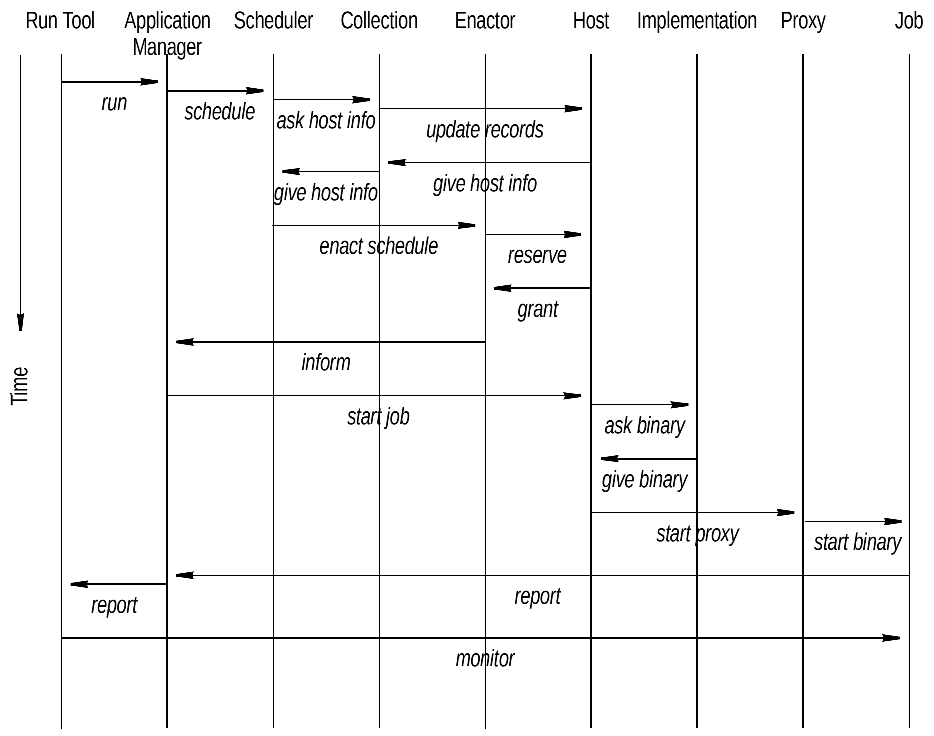

Figure 1. Scheduling Process in Legion

Third, the scheduler can be replaced. Each and every component of a Legion grid is replaceable. Thus the scheduler in the picture can be replaced by a new one that employs any algorithm of choice. Not just the scheduler, but the toolset that uses the scheduler can be changed as well. For example, we wrote a queue object that uses a similar chain of events to mimic the operation of a queuing system. Also, we wrote a parameter-space tool that can run jobs instantaneously or send them to our queue. A Legion grid can have multiple schedulers or even multiple instances of a particular scheduler. Applications can be configured to use a specific scheduler. Thus, the Legion grid resource management framework explicitly allows for different schedulers for different classes of applications. Of course, users can bypass the entire scheduling mechanism, by specifying machines directly or using some non-Legion tool for constructing a schedule for their applications. Bypassing the scheduling mechanism does not mean bypassing security and fault-tolerance, because those functions are at lower levels in the stack. Naturally, if desired, lower levels can be replaced or eliminated as well with the attendant implications.

Fourth, the scheduling infrastructure can be used as a meta-scheduling infrastructure as well. The host object shown in Figure 1 could be running on the front-end of a queuing system or the master node of an MPI cluster. Thus, Legion could be used to select such a host, but subsequent scheduling on the queue or the cluster could be delegated to the queuing system or the MPI system.

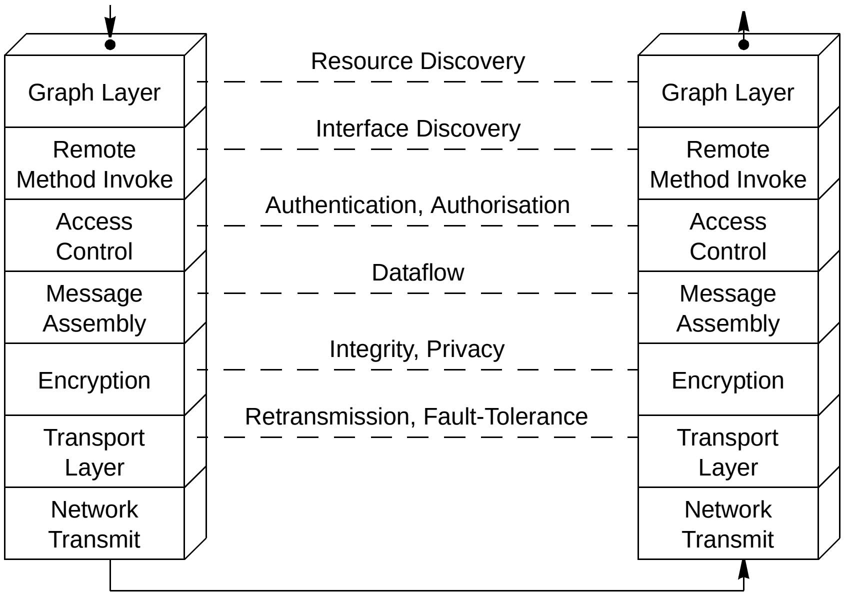

Figure 2. Protocol Stack in Legion

Fifth and perhaps most importantly, Figure 1 and Figure 2 are not restricted to scheduling alone. This comment touches on the core of the Legion philosophy. Since the definition of a resource is not restricted to CPUs alone, but can include applications, files, databases, users, etc. a similar figure can be constructed for any chain of events in Legion, representing various actions on a grid. For example, even if we consider non-scheduling tasks, such as replicating data, notifying administrators of failures or novel methods for collaboration on a grid [NAT01A], the same protocol stack applies, with the attendant benefits of security, fault-tolerance, etc. Thus in Legion, general resource management is conducted using a common framework.

When designing the Legion grid resource management framework, we had a wider definition of resource management than most other distributed systems. We tried to construct a framework within which other parties could write schedulers for different classes of applications. We consciously did not design for only the “classic” applications – long-running, compute-intensive, parallel applications, requiring high performance. Naturally, we did provide a single reference implementation of a scheduler in order to perform resource management on a Legion grid immediately upon installation. However, we intended this scheduler to be a default – a “catch-all” scheduler for users who wished to use a Legion grid as-is. We always intended permitting other schedulers to be part of any Legion grid. The resource management framework in Legion reflects several of the design principles that guided Legion:

- no change to underlying systems or network
- trade-offs between level and cost of service
- negotiation between job requirements and site autonomy

- legacy application support
- support for diversity

and so on. As we discuss in the next section, we did make a few mistakes in the implementation; however, we continue to believe that the original design is the right approach for grids. Since the lessons learnt from our experience in Legion can be generalised to the design of any grid resource management framework, analysing them can be instructive for future work.

3. Lessons Learnt from Legion

This article is about the lessons we learnt and observations we made about grid resource management. We did make some poor choices in the design of our grid infrastructure; some of them in the scheduling framework. The limitations of some of these choices were technical, whereas others were psychological. If we were to re-design Legion, here are some more lessons we would keep in mind.

Obs. 8 *People are reluctant to write schedulers for new frameworks.*

In hindsight, this lesson is unsurprising. We cannot and should not expect people to construct support for new software. A software infrastructure must gestate and mature for a while before enthusiastic and talented outsiders can contribute to it. What the computer science community has seen of Linux, Perl, GNU, etc. will be true of grids as well – initially, a small group of dedicated and talented persons create the software, and much later the pool of contributors widens drastically. Initially, we expected grid scheduling experts to write schedulers for Legion; we still hope they will write. However, we cannot rely on them to write us a scheduler. In turn, we must provide at least a default scheduler and perhaps an interesting suite of schedulers for some classes of applications. Once we learnt this lesson, we wrote two new schedulers to complement the default scheduler that already came with every Legion installation. One was a round-robin scheduler for creating instances of files, directories and other objects on the grid. The round-robin scheduler made quick decisions based on a machine file that was part of its state, thus avoiding expensive scheduling decisions for simple object creation (see Obs. 10). The second scheduler was a performance-based scheduler for parameter-space studies. This scheduler took CPU speeds, number of CPUs and loads into account for choosing machines on which to run parameter-space jobs.

Obs. 9 *Writing schedulers deep into a framework is hard.*

While we did provide a framework for writing schedulers, a mistake we made was requiring scheduler writers to know too much about Legion internals. Typically, in addition to the scheduling algorithm of interest, a scheduler writer would have to know about schedulers, enactors, hosts, classes and collections; their internal data structures; the data they packed on the wire for several method calls; and Legion program graphs. The effort required to write such a “deep scheduler” was too much to encourage rapid development. In essence, we had violated one of our own principles: ease-of-use. Our mistake lay in making Legion easy-to-use for end-users, but not necessarily so for developers. Once we recognised our error, we wrote a “shallow scheduler”, i.e., a scheduler that was about as complex as the default scheduler but did not require knowing too much about Legion internals. The performance-based scheduler for parameter-space studies mentioned earlier in Obs. 8 is an example of a shallow scheduler. This scheduler is a self-contained Perl script that requires knowing about one database of attributes (called the collection object) and the one command to access it. Not having to know Legion details was a significant advantage for this scheduler: where the deep scheduler required understanding 6600 lines of C code spread over 9 source code and header files, the shallow scheduler required understanding 800 lines of Perl code contained in one file. In addition, the deep scheduler came with no documentation other than in-source comments, whereas the shallow scheduler contained usage and help as well.

The lesson we learnt from this experience was that a high cost of constructing new schedulers is a deterrent to development. Another lesson we learnt was that a high cost of running a scheduler can hurt a grid as well. Put differently, we learnt that a quick and acceptable scheduler is much better than a slow but thorough scheduler.

Obs. 10 *High scheduler costs can undermine the benefits of scheduling.*

In Legion, a scheduler is invoked every time an object must be placed on some machine on a grid. Typical object placements involve placing jobs, or instances of applications, on compute resources. However, given the Legion view of scheduling as a task for placing any object not just a compute object, creating files and directories, implementations and queue services, consoles and classes, all require an intermediate scheduling step. For long, the scheduler that would be invoked for any creation was the default scheduler. While we fully understood the need for different schedulers for different kinds of objects, an artifact of our implementation was that we created only one scheduler – the default one.

The default scheduler’s algorithm was complex in two respects. One, the actual processing time took long, especially as the number of machines in a grid grew. Moreover, the scheduler constructed alternative or variant schedules for every request just in case the first schedule created did not meet with success. Two, the process invoked methods on too many remote objects. Each method call (or outcall) was a relatively expensive operation. Therefore, even a simple schedule would take too long to generate. For applications such as parameter-space studies, we often needed to create 50-100 instances of an application quickly. Further, when each instance finished, it would write result files back to the grid, which required scheduling operations again. The slow scheduler was undermining our benefits. Accordingly, we built faster schedulers which perhaps did not find near-optimal and variant schedules, but were far quicker than the default. The round-robin scheduler (see Obs. 8) made fewer outcalls and had a terribly simple algorithm for choosing hosts – it merely picked the next in its list. However, this simple scheduler was adequately-suited for scheduling files and directories. Likewise, the shallow scheduler we wrote for performance-based scheduling (see Obs. 8 and Obs. 9) scheduled parameter-space jobs quickly [NAT02]. It initially spent a few seconds building a schedule, but after that re-used the schedule for the duration of the application.

Obs. 11 *Over-complex schedulers are unnecessary.*

While we were correct in recognising the need for a framework for schedulers, we erred in making the framework too complex. The distinction is fine: all through this article we have argued and continue to argue for increased complexity in the *functionality* of resource management system (use different metrics and algorithms for scheduling different classes of applications; incorporate security, fault-tolerance, auditing, etc.; perform non-scheduling management for other classes of resources), but here we argue for reduced complexity in the *implementation* of such a scheduler. In blunt words, a scheduler should be sophisticated enough to do different things for different applications, but simple enough in implementation that it does not become a bottleneck, a critical resource or a development nightmare itself.

In Legion, we created a sophisticated scheduling framework (bolstering our theme throughout this article), but we also implemented this framework in a complicated manner (resulting in our learning the lesson here). In particular, splitting the scheduling process from the reservation process (the scheduler and enactor objects respectively), was overkill. The added flexibility this split gave us was never used, and we believe that it will not be used for a while because complex scheduling techniques, such as co-scheduling, that require reservations are useful for a small subset of applications only (see Obs. 7). Too many objects were involved in the scheduling process, making it feel like the process had too many “moving parts”. The failure of any one object could derail the scheduling process, making it hard to create new objects – files, directories, implementations, jobs, etc. – on a grid. For all this complexity, we did not envisage a way to compose schedulers. For some applications, it may be necessary to go through two or more scheduling algorithms to find a set of machines on which the application can be run. In effect, the first scheduler “filters” the set of available machines according to its own algorithm and passes the filtered set to the next scheduler. The next scheduler also filters the set of machines it receives according to its algorithm and passes a new filtered set to the next scheduler in line and so on. The filtered set output by the last scheduler is the schedule used for the application. Composing schedulers in this manner can enable performing complex scheduling as a series of filtering operations. Such scheduling may be required for applications that have minimum CPU requirements plus minimum disk requirements plus tool requirements plus licence requirements and so on. The requirement for such a framework is likely to arise in the future.

Obs. 12 *Clear, timely, obvious error propagation is essential*.

The lesson of clear error propagation in times of failure is obvious in principle, but ignored often in practice. Often error indicators are buried too deep in software to be useful to an end-user. In distributed systems, the problem is amplified because failures may occur on machines remote to the user. Leslie Lamport has complained that “a distributed system is one on which I can do no work because a processor I’ve never heard of has crashed.”* A grid, like any distributed system, can be notoriously hard to debug. Clear, timely and obvious error propagation is essential in order to make a grid palatable to an end-user. In our scheduling framework, we neglected to be rigorous in our error propagation. As a result, users’ jobs sometimes would fail to schedule without giving clear reasons why. We have improved error propagation significantly and continue to be diligent about eliminating this drawback.

Observations on Grid Resource Management

1. Grid Resources = CPUs + Disk + Memory + Files + Databases + Users + Administrators + Applications + Licences + Running Jobs + Mobile Devices + …
2. Grid Resource Management = Grid Scheduling + Security + Fault-tolerance + Site Autonomy + Licensing + Auditing
3. Grid Scheduling = Optimisation_Function(Throughput, Latency, Cost, Deadlines, Fairness, …)
4. Grid Applications = Parallel Applications + Sequential Applications + Pleasingly-Parallel Applications + Transactions + Soft Real-Time Applications + …
5. Grid Applications = Long-Running Compute-Intensive Processes + Large Collections of Short Jobs + Frequent Short Transactions + …
6. Scheduling Mechanism > Queuing Systems
7. Scheduling is a Negotiation between Resource Providers and Consumers
8. People are reluctant to write schedulers for new frameworks.
9. Writing schedulers deep into a framework is hard.
10. High scheduler costs can undermine the benefits of scheduling.
11. Over-complex schedulers are unnecessary.
12. Clear, timely, obvious error propagation is essential.

4. Summary

We shared some of the lessons we learnt from our experiences with grid resource management in Legion. Several of these lessons dispel long-held myths about what grid resource management means. In particular, we would like to remind readers that grid resource management is more than just co-scheduling long-running parallel jobs on queuing systems. As grids mature, diverse resources will be included in grids and grid resource management will be central to the working of a grid. We hope that the lessons presented here will serve to focus attention on important classes of applications that require resource management as well as serve to guide the design of resource managers. In particular, we believe that the pressing challenges that face the grid community are the

* The quote in the text seems to be the most popular variant. A variant with citation information is “A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”, Leslie Lamport, as quoted in CACM, June 1992.

design of rich and flexible resource specification languages in order to match resources with requests, and the design of a framework that can incorporate different solutions for different aspects of grid resource management.

ACKNOWLEDGEMENTS

We thank John Karpovich for reading earlier versions of this article and recommending several changes that improved the content and tone of this article. We thank the members of the Legion Team at the University of Virginia for their hard work over the years.

This work was supported in part by the National Science Foundation grant EIA-9974968, DoD/Logicon contract 979103 (DAHC94-96-C-0008), the NASA Information Power Grid program, and NPACI, National Science Foundation grant ANI-0222571 and National Science Foundation grant ACI-0203960.

5. References

ABR95 Abramson, D., Sosic, R., Giddy, J., Hall, B., *Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations*, 4th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Aug. 1995.

ALT90 Altschul, S. F., Gish, W., Miller, W., Myers, E. W., Lipman, D. J., *Basic local alignment search tool*, Jour. Mol. Biol., 215(3):403-410, Oct. 1990.

ARN01 Arnold, D. C., Vadhiyar, S., Dongarra, J., *On the Convergence of Computational and Data Grids*, Par. Proc. Let., 11(2-3):187-202, Sep. 2001.

BAY99 Bayucan, A., Henderson, R. L., Lesiak, C., Mann, N., Proett, T., Tweten, D., *Portable Batch System: External Reference Specification*, Tech. Rep., MRJ Technology Solutions, Nov. 1999.

BER96 Berman, F., Wolski, R., *Scheduling from the perspective of the application*, 5th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Aug. 1996.

BER99 Berman, F., *High-Performance Schedulers*, Chapter 12, The GRID: Blueprint for a New Computing Infrastructure, :279-309, Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

BRO83 Brooks, B. R., Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S., Karplus, M., *CHARMM: A Program for Macromolecular Energy, Minimization, and Dynamics Calculations*, J. Comp. Chem., 4, 1983.

BUY01 Buyya, R., Abramson, D., Giddy, J., *A Case for Economy Grid Architecture for Service-Oriented Grid Computing*, Intl. Par. and Dist. Processing Symp. (IPDPS), 2001.

CASA88 Casavant, T. L., Kuhl, J. G., *A taxonomy of scheduling in general-purpose distributed computing systems*, IEEE Trans. on Soft. Engg., 14(2):141-154, 1988.

CASA96 Casanova, H., Dongarra, J., *NetSolve: A Network Server for Solving Computational Science Problems*, The Intl. Jour. of Supercomputer App. and High Perf. Computing, 1997.

CHAP99 Chapin, S. J., Katramatos, D., Karpovich, J. F., Grimshaw, A. S., *Resource Management in Legion*, Future Generation Computing Syst., 15:583-594, Oct. 1999.

CZAJ01 Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C., *Grid Information Services for Distributed Resource Sharing*, 10th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Aug. 2001.

EDW03 Edwards, G., —, Personal Commn., Celera Genomics, Feb. 2003.

FERR98 Ferrari, A. J., Grimshaw, A. S., *Basic Fortran Support in Legion*, Tech. Rep. CS-98-11, Univ. of Virginia, Mar. 1998.

FERR99 Ferrari, A. J., Knabe, F., Humphrey, M. A., Chapin, S. J., Grimshaw, A. S., *A Flexible Security System for Metacomputing Environments*, High Perf. Computing and Networking Europe (HPCN), Apr. 1999.

FOS99 Foster, I., Kesselman, C., *The Globus Toolkit*, Chapter 11, The GRID: Blueprint for a New Computing Infrastructure, :259-278, Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

FOX99 Fox, G. C., Furmanski, W., *High-Performance Commodity Computing*, Chapter 10, The GRID: Blueprint for a New Computing Infrastructure, :237-255, Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

FREY02 Frey, J., Tannenbaum, T., Foster, I., Livny, M., Tuecke, S., *Condor-G: A Computation Management Agent for Multi-Institutional Grids*, Cluster Computing, 5(3):237-246, 2002.

GAN99 Gannon, D., Grimshaw, A. S., *Object-Based Approaches*, Chapter 9, The GRID: Blueprint for a New Computing Infrastructure, :205-236, Morgan Kaufmann, ISBN 1-55860-475-8, 1999.

GEI98 Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1998.

GRIM94 Grimshaw, A. S., Wulf, W. A., French, J. C., Weaver, A. C., Reynolds, P. F. Jr., *Legion: The Next Logical Step Toward a Nationwide Virtual Computer*, Tech. Rep. CS-94-21, Univ. of Virginia, Jun. 1994.

GRIM96 Grimshaw, A. S., Ferrari, A. J., West, E., *Mentat*, Parallel Programming Using C++, MIT Press, 1996.

GRIM97 Grimshaw, A. S., Wulf, W. A., *The Legion Vision of a Worldwide Virtual Computer*, Comm. of the ACM, 40(1), Jan. 1997.

GRIM98A Grimshaw, A. S., Lewis, M. J., Ferrari, A. J., Karpovich, J. F., *Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems*, Tech. Rep. CS-98-12, Univ. of Virginia, Jun. 1998.

GRIM98B Grimshaw, A. S., Ferrari, A. J., Lindahl, G., Holcomb, K., *Metasystems*, Comm. of the ACM, 41(11), Nov. 1998.

GRIM99 Grimshaw, A. S., Ferrari, A. J., Knabe, F., Humphrey, M. A., *Wide-Area Computing: Resource Sharing on a Large Scale*, IEEE Computer, 32(5), May 1999.

GRO96	Gropp, W., Lusk, E., Doss, N., Skjellum, A., *A High-Performance, Portable Implementation of the Message Passing Interface Standard*, Par. Computing, 22(6), Sep. 1996.
HEM99	Hempel, R., Walker, D. W., *The Emergence of the MPI Message Passing Standard for Parallel Computing*, Comp. Stds. and Interfaces, 7, 1999.
HUM01	Humphrey, M. A., Thompson, M. R., *Security Implications of Typical Grid Computing Usage Scenarios*, 10th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Aug. 2001.
IBM93	International Business Machines Corporation, *IBM LoadLeveler: User's Guide*, Sep. 1993.
KARP96	Karpovich, J. F., *Support for Object Placement in Wide Area Distributed Systems*, Tech. Rep. CS-96-03, Univ. of Virginia, Jan. 1996.
KEN02	Kennedy, K., Mazina, M., Mellor-Crummey, J., Cooper, K., Torczon, L., Berman, F., Chien, A., Dail, H., Sievert, O., Angulo, D., Foster, I., Aydt, R., Reed, D., Gannon, D., Dongarra, J., Vadhiyar, S., Johnsson, L., Kesselman, C., Wolski, R., *Toward a Framework for Preparing and Executing Adaptive Grid Programs*, Intl. Par. and Dist. Processing Symp. (IPDPS), Apr. 2002.
KING92	Kingsbury, B. A., *The Network Queueing System (NQS)*, Tech. Rep., Sterling Software, 1992.
LEW03	Lewis, M. J., Ferrari, A. J., Humphrey, M. A., Karpovich, J. F., Morgan, M. M., Natrajan, A., Nguyen-Tuong, A., Wasson, G. S., Grimshaw, A. S., *Support for Extensibility and Site Autonomy in the Legion Grid System Object Model*, Journal of Par. and Dist. Computing, ?(?):?-?, ? 2003.
LIT88	Litzkow, M. J., Livny, M., Mutka, M. W., *Condor – A Hunter of Idle Workstations*, 8th Intl. Conf. of Dist. Computing Syst., :104-111, Jun. 1988.
LIU02	Liu, C., Yang, L., Foster, I., Angulo, D., *Design and Evaluation of a Resource Selection Framework for Grid Applications*, 11th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Jul. 2002.
LIV99	Livny, M., Raman, R., *High-Throughput Resource Management*, Chapter 13, The GRID: Blueprint for a New Computing Infrastructure, :311-337, Morgan Kaufmann, ISBN 1-55860-475-8, 1999.
MAC98	MacKerell, A. D.. Jr., Brooks, B. R., Brooks, C. L. III, Nilsson, L., Roux, B., Won, Y., Karplus, M., *CHARMM: The Energy Function and Its Parameterization with an Overview of the Program*, The Encycl. of Comp. Chem., 1, 1998.
NAK99	Nakada, H., Sato, M., Sekiguchi, S., *Design and Implementations of Ninf: towards a Global Computing Infrastructure*, Future Generation Computer Syst., Metacomputing Issue, 15(5-6):649-658, Elsevier Science, 1999.
NAT01A	Natrajan, A., Humphrey, M. A., Grimshaw, A. S., *Capacity and Capability Computing in Legion*, The 2001 Intl. Conf. on Computational Sc. (ICCS), :273-283, May 2001.

NAT01B	Natrajan, A., Humphrey, M. A., Grimshaw, A. S., *Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context*, High Perf. Computing Syst. (HPCS), Jun. 2001.
NAT01C	Natrajan, A., Crowley, M., Wilkins-Diehr, N., Humphrey, M. A., Fox, A. D., Grimshaw, A. S., Brooks, C. L. III, *Studying Protein Folding on the Grid: Experiences using CHARMM on NPACI Resources under Legion*, 10th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Aug. 2001.
NAT02	Natrajan, A., Humphrey, M. A., Grimshaw, A. S., *The Legion Support for Advanced Parameter-Space Studies on a Grid*, Future Generation Computer Syst., 18(8):1033-1052, Elsevier Science, Oct. 2002.
NGU00	Nguyen-Tuong, A., *Integrating Fault-tolerance Techniques in Grid Applications*, Ph.D. Diss. CS-2000-05, Univ. of Virginia, Aug. 2000.
RAM98	Raman, R., Livny, M., Solomon, M. H., *Matchmaking: Distributed Resource Management for High Throughput Computing*, 7th IEEE Intl. Symp. on High-Perf. Dist. Computing (HPDC), Jul. 1998.
SEIG96	Seigel, J., *CORBA Fundamentals and Programming*, Wiley, ISBN: 0471-12148-7, 1996.
SMI00	Smith, W., Foster, I., Taylor, V., *Scheduling with Advanced Reservations,* Intl. Par. and Dist. Processing Symp. (IPDPS), May 2000.
SMI03	Smith, C., —, Personal Commn., Platform Computing, Feb. 2003.
SNIR98	Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., *MPI: The Complete Reference*, MIT Press, 1998.
WEI95	Weissman, J., *Scheduling Parallel Computations in a Heterogeneous Environment*, Ph.D. Diss. CS-1995-06, Univ. of Virginia, Aug. 1995.
ZHOU92	Zhou, S., *LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems*, Work. on Cluster Computing, Dec. 1992.
ZHOU93	Zhou, S., Wang, J., Zheng, X., Delisle, P., *Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems*, Soft. Prac. and Exp., 23(2), 1993.