University of Alberta

PLACEHOLDER SCHEDULING FOR OVERLAY METACOMPUTING

by

Christopher James Pinchak

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2003

Abstract

The availability of a large number of distributed high-performance computing resources has given rise to the field of metacomputing in which these resources are coupled to obtain their combined benefits. Placeholder scheduling and TrellisWeb were developed to provide a metacomputing environment. Placeholder scheduling takes advantage of existing software infrastructure to minimize problems introduced by administrative boundaries and provide performance benefits to users. Existing batch schedulers are not replaced by placeholder scheduling. Placeholders are instead layered on top of existing schedulers to take advantage of the local policies they represent. Experimental results show that placeholder scheduling scales to a large number of sites and administrative domains, and can dynamically load balance jobs across all participating sites in such a way that the makespan is significantly lower than with an alternative method.

Acknowledgements

Seldom is any endeavour the undertaking of a single individual, and this thesis is no exception. Many people played many parts during the research described herein, and without their help this thesis would not exist. For research to be successful, someone must have the ability to judge what is reasonable and what is not. To this end, I credit my supervisor, Dr. Paul Lu. Paul gave me ideas when I had none, motivation when things looked bleak, and assurances that I was on the right track. Paul, along with Dr. Jonathan Schaeffer, conceived of the CISS project and, through their determination, set it in motion, providing me with an opportunity to showcase my work in the public eye. This was a unique experience that I will always remember.

Also important are those involved in a less technical sense. Lesley Schimanski helped greatly by always being supportive of my goals, in addition to her endless proof-reading of this thesis. It would not be nearly the work that it is without her help. Also important are my parents, Bob and Sherry Pinchak, for always being excited and interested in my continuing education.

Lastly, I would like to thank the participants of CISS-1. The hours leading up to and during the experiment were fraught with minor glitches, and many of you went beyond the call of duty to ensure that our big day went as smoothly as possible. This thesis is as much a testament of your hard work as it is mine.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ever-increasing demand for more powerful high-performance computers has spurred the development of *metacomputing* (and the closely-related area of *grid computing*). The primary goal of a metacomputing environment is to provide the illusion that the individual member sites, together, are a single resource. Metacomputers are a subset of computational grids and are more limited, by design, with respect to the environment they create (e.g., they provide no support for grid-enabled devices such as those used for data collection or visualization). Without a metacomputing environment, users view individual sites as just that – individual sites. Each site requires its own login and has its own policies that users must adhere to. One of the sites may be a local or preferred site for the user, perhaps where the user develops software or runs tests. The others are seen as remote sites, and often do not have the same programs and data that the preferred site does. In the common case, the preferred site is chosen for the majority of application runs, with remote sites being seldom used. Although hardware continues to make performance leaps and bounds, users are beginning to demand more computing resources than are currently available. This demand, coupled with current high-bandwidth networks, has led to many efforts aimed at amalgamating multiple sites into a single virtual supercomputer.

Users do not need to know or care where their applications execute within a metacomputer, and would rather focus on more important tasks such as preparing input and examining output. Issues such as the local scheduler interface, data file locations, and machine load can become overwhelming to even the most experienced user should they have access to a large number of different sites. Simply put, such issues are time consuming unless some sort of metacomputing environment is in place to deal with them.

This work describes placeholder scheduling and its role in a metacomputing environment. Placeholder scheduling differs from other metacomputing and grid computing software, such as Globus [8, 9] or Legion [4, 13], in that it is not designed to be a comprehensive solution. For example, this work ignores the legitimate design goal of resource discovery, in which new resources are automatically located by the system and incorporated into the metacomputer dynamically. Rather, it focusses on the assignment of jobs to individual sites in an efficient way that is semi-transparent to the user.

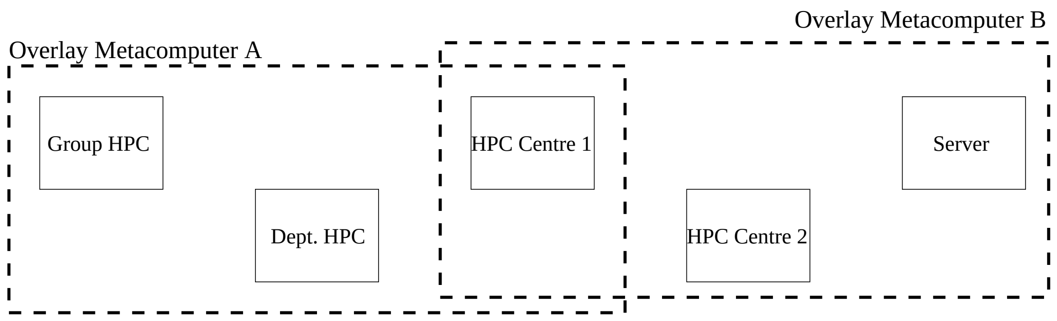


Figure 1.1: Overlay Metacomputers

The user explicitly defines personal *overlay metacomputers* that they may use to execute jobs. For example, in Figure 1.1, Researcher A has access to his group's system, a departmental system, and a system at a high-performance computing centre. Researcher B has access to her group's server and two different high-performance computing centres, including one in common with Researcher A (HPC Centre 1). With the use of overlay metacomputers, Researcher A creates Overlay Metacomputer A (containing HPC Centre 1) and Researcher B creates Overlay Metacomputer B (also containing HPC Centre 1). Now, both Researcher A and Researcher B may make use of all systems they have access to.

Placeholder scheduling was designed to coexist with local site policies, including batch scheduling systems, rather than replace or circumvent them. To avoid policy conflicts, placeholder scheduling requires as little as possible from sites participating in the overlay metacomputers. For example, if a user wants to use Site A as part of an overlay metacomputer, all that is required by Site A is the provision of a user account. Placeholder scheduling is not intended to provide a metacomputer suitable for all possible situations, but instead deals with many of the practical issues involved in creating one that incorporates some level of heterogeneity.

1.1 Goals of a Metacomputing System

Because metacomputers are created from multiple computers at multiple sites, the computers involved are often under the control of more than one entity (e.g., universities). Each entity has its own team of system administrators that set local policy, and these local policies may be incompatible with each other due to choices made at each site (e.g., using one batch scheduler instead of another). Each of these entities represents a unique *administrative domain* within the metacomputer, and must be dealt with according to the local policies set by the system administrators.

An ideal metacomputing system would create a single, seamless virtual computer from a collection of physical computers. This seamless virtual computer would be manipulated as a single machine and would provide the aggregate resources of all component computers to the user. The user could submit jobs to the metacomputer without regard to where the resources are physically located. A comprehensive computational grid is the goal of some existing projects [8, 13].

However, some difficult issues arise when building a metacomputing system. Briefly, they are:

1. **Resource Scheduling & Allocation**: Because there are potentially a large number of individual underlying systems, a resource scheduling scheme must exist that decides which resources will be used for which jobs. The primary resource of interest is CPU time. When a single administrative domain is involved, this is usually the responsibility of the batch scheduler. Batch schedulers are capable of providing a metacomputing environment of sorts through the use of routing queues (such as those in the Portable Batch System [25]) that are able to forward jobs to other computer-specific queues that the scheduler controls. The resource allocation policy may require complete control of the underlying resources (meaning access is only allowed from the metacomputing system), or it may exist on top of local policies and systems. Finally, the resource allocation policy may consider any combination of resources (CPU, memory, and data file location, for example) when making scheduling decisions.

2. **Single Log On**: For the metacomputing system to provide the illusion of a single virtual computer, the user must be able to log onto the single computer as if it is real. This passes the responsibility of executing actions on behalf of the user onto the metacomputing system itself. The user should not be required to provide authentication credentials (such as a password or private key, for example) each time the metacomputing system performs some action on his or her behalf. Some potential solutions for providing a single metacomputer identity are to execute all commands under a special, well-known account on each system, or to allow the user to designate credentials to be used by the metacomputing system.

3. **Access Control**: An important part of any operating system is controlling the access permissions of users. A fundamental question that must be asked by an operating system is “Is this user allowed to access this computer in this way?” Like an operating system, a metacomputing environment must provide some level of access control. Some metacomputer users may not be permitted to use some machines, and the metacomputing system must respect this. Ultimately, local policies must be specified and respected so that participating sites maintain control of their resources.

4. **Data Access**: Data files are an important part of most computations. As such, their availability on a system prior to execution of the computation is crucial to the success of that computation. Operating systems solve this problem by using a filesystem that provides a global namespace for all files that may be used by applications. Operating systems also provide the ability to cross-mount filesystems, thereby allowing multiple computers to share a single filesystem. However, in the absence of a single administrative domain, such cross-mounting of filesystems

is usually not allowed. Therefore, global file access must be provided by other means in a metacomputing system.

In order for the metacomputing system to be usable, some subset of the above concerns must be addressed. Each of the domains is an individual subject of research, and a metacomputing system that addresses all domains to the satisfaction of all involved is still a number of years away.

1.2 Goals of Global Scheduling

A mechanism for metacomputing should achieve the following goals with respect to global resource scheduling and allocation:

1. **Handle Multiple Administrative Domains**: Although some metacomputers may exist within the confines of a single administrative domain, most will not be as restricted. Crossing administrative boundaries is a reality that must be addressed by a modern metacomputing system because users wish to make use of an increasing number of diverse resources. Different administrative domains mean different local policies, such as choice of local batch scheduler and use of account quotas. A global scheduler must be able to make use of resources from a wide variety of administrative domains, and should do so while adhering to the local policies set out by the administration.

2. **Batch Scheduler Interaction**: Many high-performance computing sites employ one or more batch schedulers to govern access to local computational resources. A global scheduler should not have to replace underlying batch schedulers, nor should it create a new one. Rather, a global scheduler should take advantage of the underlying batch schedulers by layering on top of them.

3. **Performance Benefits**: A global scheduler is expected to provide a performance benefit to users. This performance benefit comes from the ability of users to make use of more resources than they would be able to without a global scheduler. Without a meaningful performance gain, users would not use the system. Ideally, a global scheduler should provide lower makespans for a user workload than any alternative methods the user has available to them.

1.3 Features of Placeholder Scheduling and TrellisWeb

Placeholder scheduling is a mechanism for metacomputing in which each placeholder represents a potential unit of work. Some of the parameters of a placeholder (such as the executable name) may be omitted from a placeholder when it is submitted to a computing system. When the placeholder begins executing, it contacts a central authority and requests the missing parameters. In this way, parameters are *pulled* by the placeholder rather than *pushed* by the central authority. Placeholder scheduling is described in detail in Chapter 3.

Placeholder scheduling and TrellisWeb were designed to address some of the problem domains just described. TrellisWeb is a continuation of the PBSWeb system [20, 21] that was originally designed to simplify job submission to the PBS batch scheduling system. Specifically, PBSWeb was designed to:

1. Provide a portal capable of interacting with PBS systems on behalf of the user.
2. Allow the user to interact with multiple such systems.
3. Make such interactions as easy as possible by simplifying the interface and recording past settings so that they may be reused.

PBSWeb could be adapted to handle multiple batch scheduling systems in addition to PBS. These goals are now also part of TrellisWeb.

TrellisWeb provides a convenient interface to placeholder scheduling. TrellisWeb and placeholder scheduling combine to address three of the four above goals of a metacomputing environment in the following ways:

1. **Resource Scheduling & Allocation**: Placeholder scheduling makes use of the underlying scheduling systems already in place at participating sites. This allows local users access to the computer whether or not they are using placeholder scheduling. Furthermore, layering on top of the existing scheduling systems allows each site to specify and enforce its own local scheduling policy. Such policies may include, for example, a static or dynamic priority scheme, and queue structure. Placeholders are only allowed to access the batch schedulers at the user level, and do not have privileged access to the scheduler. Additionally, placeholder scheduling may take advantage of sites without underlying batch schedulers (zero-infrastructure sites) by using special placeholders that assume unrestricted use of the site. More details are provided in Chapter 3.
2. **Single Log On**: TrellisWeb provides a web-based graphical user interface (GUI), or portal, with a username/password log on. Furthermore, TrellisWeb performs all actions at sites on behalf of the user utilizing the user's account. This is achieved through existing Secure Shell (SSH) [3] infrastructure that must be properly configured prior to using TrellisWeb (possibly, but not necessarily, with the assistance of system administrators). TrellisWeb transparently accesses an authenticated user's accounts at all sites such that the user is not prompted multiple times for a password.
3. **Access Control**: Placeholder scheduling and TrellisWeb use existing user accounts to provide a metacomputing environment. As a result, placeholder scheduling and TrellisWeb may only use systems that the user has accounts for. Prior negotiation is required between the user and system administrators during which site policy is consulted to determine whether the

user should be granted access. Furthermore, placeholder scheduling and TrellisWeb are only capable of performing actions that the user would be able to perform manually. No attempts are made to circumvent user-level security policies in force at various sites. Therefore, the local site policies are reflected implicitly within placeholder scheduling and TrellisWeb.

Placeholder scheduling does not address the domain of file access, as the primary goal of placeholder scheduling is job scheduling. However, another application under the banner of the Trellis project at the University of Alberta [32], called the Trellis File System [29], addresses this particular domain exclusively.

1.4 Overview of Placeholder Scheduling and TrellisWeb

Placeholder scheduling concentrates mainly on global scheduling issues, such as load balancing and makespans for workloads. These particular criteria were chosen because of the observation that loads are often unbalanced across multiple high performance computing sites. One possible explanation for this imbalance is that users select favourite or preferred sites to perform the majority of their computations. A preferred site may be specially configured and may contain the majority of the data files required by the computation. For whatever reasons, the barriers to using computers at other sites can cause the sites preferred by a large user base to become heavily used.

To combat the problem of load imbalance, placeholder scheduling abstracts away the individual computers with the use of an overlay metacomputer. Users create an overlay metacomputer from a list of computers they have access to. Although the overlay metacomputers may be as restrictive as desired (they may be composed of as little as one physical computer), the greatest benefit is obtained from using as many individual computers as possible. More than one overlay metacomputer may exist at a time, and the user may make each overlay metacomputer as general or specific as desired. The primary idea behind overlay metacomputers is to specify them on a per-application basis to ensure that all subsystems of the overlay metacomputer are capable of executing the same application.

Placeholders do not require any special software, other than SSH, to be installed on individual computers. Moreover, they utilize existing normal user accounts at various sites, thereby allowing the sites to enforce per user policies such as disk quotas and access restrictions. Because no special privileges are required to use an overlay metacomputer, we consider overlay metacomputers created with TrellisWeb to be *user-level* overlay metacomputers.

1.5 Concluding Remarks

A metacomputing system is charged with the task of providing access to multiple distributed resources. This environment should also address other concerns such as single log on, access control, security, and data access. Placeholder scheduling and TrellisWeb are able to address many of these concerns in a simple, straight-forward manner without requiring an excessive amount of control or additional software. TrellisWeb is built on existing software that was designed to make access to batch scheduled systems easier for the user, and has the additional requirement of providing access to multiple resources to form a metacomputing environment. Through the use of placeholder scheduling, a user-specified workload may be balanced across multiple sites so as to provide a noticeable decrease in execution time (makespan). The key feature of placeholder scheduling and TrellisWeb is that they obtain a number of benefits while making few sacrifices.

Chapter 2

Related Work

Placeholder scheduling is not the first project that attempts to create a metacomputing environment. A number of projects have addressed problems related to metacomputing, some of which deal with more fundamental problems than others. In the following sections, some closely-related metacomputing projects are discussed.

Placeholder scheduling and TrellisWeb focus on solving a subset of the problems addressed by other projects. Many projects overlap on some of the problems that they address, but each have unique advantages and disadvantages. The primary advantages and current disadvantages are briefly listed in Table 2.1, and the systems themselves are discussed in the remainder of the chapter.

2.1 Globus

The goal of the Globus project [8, 9] is to provide a toolkit sufficient for building a computational grid. A computational grid is similar to a power grid in which computational cycles are traded like electricity. Globus represents one of the largest and most mature grid projects, and the toolkit contains elements that address a large number of issues that could arise when building a grid. The Globus toolkit is a comprehensive set of software components that includes:

1. Resource management: Globus Resource Allocation Manager (GRAM)
2. Interprocess communication: Nexus
3. Security: Globus Security Infrastructure (GSI)
4. Information service: Metacomputing Directory Service (MDS)
5. Health and status monitoring: Heartbeat Monitor (HBM)
6. Remote data access: Globus Access to Secondary Storage (GASS) (now deprecated)

The goal of the Globus project is to develop standardized components from which grids can be built. Each component of the toolkit may be used individually, and a grid developer may use any or

System	Brief Description	Main Advantages	Current Disadvantages
Globus	A comprehensive grid toolkit.	Comprehensive: Addresses a wide range of metacomputing issues including heterogeneous environments. Standards-based: Working toward standardized grid components.	Requires the installation and maintenance of a great deal of additional infrastructure.
Legion	An object-oriented metacomputing operating system.	Object-Oriented: Provides a standardized interface because everything is an object. Wide-Area Operating System: Provides a seamless operating system over the entire object space (including all hosts).	Requires all applications to be cast in the object-oriented framework of Legion. All sites participating in a Legion metacomputer must run Legion software. Scheduling is reservation-based, which may not be easily supported by some sites.
EveryWare	An application-level grid toolkit.	Allows the use of resources at the meta-metacomputing level, including batch schedulers and grid-controlled resources (e.g., Globus).	Applications must be integrated with the toolkit to take advantage of it.
Batch Schedulers	Systems designed to queue and schedule jobs at high-performance computing sites.	Scheduling: The system is able to make scheduling decisions for submitted applications. Common Policy: A fixed number of policies may be implemented for all participating sites.	May not work well in multiple administrative domains (conventional batch schedulers). Redirects system calls back to the originating host (Condor and Condor-G).
Parameter Space Systems	Systems designed to aid in the execution of parameter space studies such as simulations.	Greatly simplifies the execution of parameter space studies from start to finish.	Restricted to applications that are run as parameter space studies. Little consideration is made for applications that do not fit this model.
Portals	GUIs created to ease the use of widespread resources such as metacomputers.	Makes access to multiple resources much easier.	Are limited to assisting in accessing multiple resources, and do not incorporate large-scale integration software.
TrellisWeb	A system that provides metacomputing via placeholder scheduling and a GUI to make placeholder scheduling easily usable.	Only a normal user account and SSH access are required to obtain the performance benefits of a metacomputer.	Not comprehensive enough to allow for the integration of all possible systems. Does not provide facilities for concurrently executing a single task on multiple sites.

Table 2.1: Advantages and Disadvantages of Related Systems

all of the toolkit components when developing a new grid. The services provided by Globus toolkit components are meant to deal with low-level issues, such as security, thereby freeing the developer to concentrate on grid functionality rather than mundane tasks such as dealing with heterogeneity within the grid.

Placeholder scheduling and TrellisWeb only deal with a subset of areas addressed by the Globus toolkit, namely those of resource management and security. Globus toolkit components are designed to be as comprehensive as possible so that a grid can be created out of almost any combination of systems imaginable, no matter how diverse they may be. As a result, Globus must be as general as possible. Placeholder scheduling is not as comprehensive, and therefore sacrifices some potential diversity for decreased complexity and lower infrastructure requirements.

2.1.1 Globus Resource Allocation

Resource allocation is performed in the Globus toolkit using Globus Resource Allocation Managers (GRAMs) [5]. GRAMs are responsible for allocating resources specified by a description written in Resource Specification Language, and do **not** perform scheduling. Rather, GRAMs are responsible for matching user requests to machines in the Grid.

The GRAM infrastructure matches requests to resources through a network of resource brokers. Each broker makes a request more specific. For example, a user may specify a requirement such as “needs at least 1 GB of storage”. The resource brokers then break this requirement down into a list of computers capable of providing the necessary 1 GB of storage space, and one is eventually selected. Resource brokers are also able to break a single request into multiple requests, thereby allowing multiple GRAMs to co-allocate resources on multiple computers.

Placeholder scheduling currently does not provide the means for a user to specify particular resource requirements that are then matched with machines. Instead, placeholder scheduling takes a simpler approach whereby a user may specify an overlay metacomputer such that all sites within the overlay metacomputer are capable of executing an application. By specifying an overlay metacomputer, no explicit resource matching needs to be performed, and the focus can instead be shifted to scheduling, something not provided by GRAM. Placeholders also schedule jobs on resources such that the makespan of a set of jobs is minimized.

TrellisWeb does not provide any mechanism for resource discovery or brokering. This is due in large part to the fact that overlay metacomputers are not expected to change a great deal in a short amount of time. Users know where they have accounts and what the capabilities of the systems are, so the burden of discovery and brokering is placed on them. By avoiding resource discovery and brokering, TrellisWeb avoids the corresponding complexity introduced when appropriate resources have to be located for jobs.

2.1.2 Globus Security

Globus security, provided by the Globus Security Infrastructure (GSI) [10], is based on the use of Globus credentials and proxies. A user proxy acts on behalf of the user, and a resource proxy acts on behalf of a site. Whenever a user wishes to access resources at a site, the user proxy must contact the resource proxy with the provided credentials (or new credentials derived from those provided). It is then up to the resource proxy to consult local policy to determine whether the user is allowed access. If so, the resource proxy provides a mapping from Globus identifier to a local identifier (i.e., local user ID).

The key feature of GSI is its ability to handle multiple types of secure operations (such as login via SSH and communication via the Secure Socket Layer). By leaving the mapping of Globus credential to local credential up to the resource proxies, each site can be dealt with on an individual basis. This provides the maximum amount of flexibility in terms of local authentication mechanisms, as a number of options may be specified at the site level.

Placeholder scheduling and TrellisWeb simply use the existing SSH infrastructure instead. Although GSI is capable of handling SSH access via the resource proxy, it is designed to be far more general. As a result, a resource proxy must exist at each site to provide authentication and mapping services. Because placeholder scheduling is designed to minimize the amount of required infrastructure, for now, it utilizes the existing SSH infrastructure. Basic SSH infrastructure is assumed, as it is already present and supported on most Unix and POSIX systems. For example, every Linux distribution comes with OpenSSH.

2.2 Legion

The Legion system [4, 13] is designed to provide a single virtual computer from a collection of physical computers. Legion is based on an object model in which everything (hosts, users, files) is an object. Legion places all resources under common control, effectively creating a wide area operating system. This is different than Globus and placeholder scheduling, in which users need not see the entire metacomputer, but simply the parts they wish to use.

To be part of a Legion metacomputer, Legion hosts must run Legion software. However, the hosts are free to define their own policies by providing an implementation of a host object. Host objects are the final authority on which applications are permitted to run on a computer. This allows Legion to accept some level of heterogeneity in the hosts.

Scheduling is performed by either user-defined or generic schedulers provided with Legion. Scheduling is intended to be performed by schedulers that are aware of the characteristics of the application. This way, a scheduler creates the best schedule for the application, but not necessarily the overall system. A scheduler may also query information about other objects, such as hosts and users, in order to make scheduling decisions. Once a scheduler has produced a schedule, it passes the schedule to an enactor that makes reservations with the host objects. Because host objects have the final say, they may accept or reject reservations made by the enactor.

Native Legion applications are written to make use of Legion system objects, although legacy applications are also supported. By requiring that Legion applications be tightly integrated with the Legion system to achieve optimal performance, the Legion system requires applications to be written in a Legion-specific format. This may be problematic should the developer of the application wish to create a widely available and easily accessible version.

A drawback of the Legion system is the requirement of writing native Legion applications for maximum advantage. At the very least, legacy applications must be wrapped in Legion objects so that they may obtain some of the Legion benefits. Each Legion application must interact with the underlying Legion system at some point. The fact that Legion files are objects is an example of one such interaction. It is often the case that users are not developing their own applications, but are purchasing them in binary form. Expecting such users to then wrap the application in a Legion object adds an extra burden on the user.

Placeholder scheduling does **not** require applications to be rewritten or modified in any way to take advantage of an overlay metacomputer. If an application is available in binary form only, the overlay metacomputer must be restricted to sites capable of executing the binary, but this is to be expected. Moreover, placeholder scheduling does not impose a model of computation akin to that of Legion objects. An application used with placeholder scheduling is simply an application that may be run with or without placeholders.

2.3 EveryWare

The EveryWare toolkit [34] provides what can be considered meta-metacomputing software. Whereas metacomputing systems seek to combine multiple computers into a single virtual resource, EveryWare goes a step further and is capable of combining multiple distinct metacomputers into a single virtual metacomputer. EveryWare integrates with particular applications and allows them to be run on systems as varied as Globus, Legion, Condor, NetSolve, Java, Windows NT, and Unix. EveryWare applications can be run across any or all of these system boundaries, effectively turning such resources into a single parallel computer. Applications are written such that they take advantage of EveryWare toolkit components for operations such as file access. The underlying EveryWare toolkit implements such operations differently depending on which system the application component executes on. By interfacing with such a variety of resources, EveryWare greatly increases the number of resources available to users and their applications.

Unfortunately, for an application to obtain EveryWare benefits it must be integrated with EveryWare software. Unlike a batch scheduling system, in which the scheduler controls the execution of the application, EveryWare must be linked with the application before it can be run. Therefore, the application itself must use services provided by the EveryWare API. Applications not explicitly written in such a way must be at least relinked with the EveryWare toolkit before they may be run, resulting in another step for the user. Moreover, such relinked applications may not run in another non-EveryWare setting.

Placeholder scheduling and TrellisWeb do not require any integration of application software and TrellisWeb software. Placeholder scheduling and TrellisWeb are more like a batch scheduler in the sense that they control the execution of the application without explicitly requiring the application to make use of additional software. Because placeholder scheduling does not provide inter-site communication beyond that already present in the application, such integration can be avoided. In many other respects, such as handling resource heterogeneity, placeholder scheduling and TrellisWeb are similar to EveryWare (except that EveryWare has demonstrated use with more diverse platforms).

2.4 Batch Scheduling Systems

Batch scheduling systems are often used as a means of controlling and limiting access to the computational resources of a computer or site. Batch schedulers are responsible for accepting requests and allocating resources that are capable of servicing such requests. Without batch schedulers, many systems would experience over-utilization as users would have to manually allocate resources to their tasks.

Some batch schedulers, such as PBS, are capable of providing a metacomputer-like abstraction through the use of routing queues. Routing queues accept jobs and then forward (or route) the jobs into a number of other queues. The routing is done with global knowledge of all jobs in all queues, and as a result good schedules can be created. However, these schedulers come at a cost. In order for the scheduling system to have global knowledge and control, all sites participating in routing queues must be controlled by the same scheduler. Likewise, all sites must run the same software. This requirement is often too strict to accommodate a diverse metacomputing environment.

2.4.1 Conventional Batch Schedulers

Conventional batch schedulers, such as the Portable Batch Scheduler (PBS) [25], Sun Grid Engine (SGE) [30], the Load Sharing Facility (LSF) [19], and others, are designed primarily to enforce resource access on high performance computers. Users submit *jobs* to the scheduler, along with a specification of the resources required to execute the job (sometimes included in the *job script*). Depending on the system, the user must also specify the *job queue* to use for the job. It is then the responsibility of the batch scheduler to locate sufficient resources to execute the job, and to begin execution when these resources become available, subject to the requirements provided by the user at job submission time.

Most conventional batch schedulers are intended to control access to a single computer or site. In general, they must be installed with superuser privileges, and execute with superuser permissions. They make scheduling decisions based on global information of all currently executing and waiting jobs. Priority schemes may be implemented to favour certain jobs over others. These properties make batch schedulers ideal for computer or site access, but batch schedulers are far too restrictive for all but the smallest metacomputing systems. Moreover, two or more differing batch scheduling systems are not able to interoperate.

Placeholder scheduling, on the other hand, is explicitly designed to schedule across multiple sites and administrative domains. Some control and precision is lost due to the fact that different administrative domains do not share scheduling information, but nonetheless, placeholder scheduling provides good performance improvements, as shown in the experiments of Chapter 5. The ideal situation is to have a batch scheduler control multiple computers from multiple sites, but when this is not possible due to administrative boundaries, placeholder scheduling is an alternative.

2.4.2 Condor

The Condor system [17, 18], although technically a cycle-stealing system (in which otherwise idle computers are used for background processing), is similar to a conventional batch scheduler. Condor was originally developed to allow a network of individual workstations to be treated as a combined pool of processors. Each workstation runs Condor software that allows it to execute waiting jobs when idle, and submit jobs to the Condor queue when requested by the user. Idle workstations execute jobs and preempt such jobs when the workstation returns to use. Condor is able to locate idle resources and migrate jobs to new sites. All system calls made by a remotely executing process are routed back to the originating machine, so the process has access to all features of the originating system. Together, the features of Condor create the illusion that the user has access to a large multiprocessor machine.

Additional work [26] has been done to allow parallel applications to execute on a Condor pool, but the applications must be sufficiently flexible to account for the opportunistic nature of Condor. Because a workstation may return to use at any time, the parallel application must be able to tolerate the sudden disappearance of one or more of the subprocesses. Although some parallel applications can tolerate dynamic resource loss (for example, parallel processes using the master/workers paradigm), many parallel applications cannot tolerate such losses (Parallel Sorting by Regular Sampling [16], for example). This makes Condor more suitable for a large number of sequential jobs rather than parallel jobs. Placeholder scheduling has no such restriction on parallel jobs, so long as the underlying sites are capable of executing the parallel application.

Condor suffers from the same restrictions as do the conventional batch schedulers. In order for a computer to take part in a Condor pool, it must run Condor software and must be placed into the Condor domain. Although the computer may safely be used interactively (and possibly by other batch schedulers), Condor was not designed to place a single computer into multiple individual pools, and includes no notion of an overlay metacomputer. In contrast, placeholder scheduling and TrellisWeb allow a single computer to be used in more than one overlay metacomputer so that users are able to define overlay metacomputers of their choosing.

2.4.3 Condor-G

Condor-G [11] builds upon the Condor system by replacing the custom resource discovery and allocation software with Globus toolkit components. Beyond that, Condor-G retains much of the functionality of the original Condor system. Condor-G is enhanced by the use of the Globus Security Infrastructure, Globus Resource Allocation Managers, the Globus Metacomputing Directory System, and Globus Access to Secondary Storage, all of which are Globus Toolkit components. These components make Condor-G a more extensive high-throughput computing system.

Condor-G uses the Globus infrastructure to obtain resources from computers that are part of the Globus grid. Condor-G provides its own scheduler, and makes use of GlideIns to perform execution on remote machines. GlideIns are simply jobs submitted through the Globus infrastructure that contact a Condor-G scheduling component to receive work when they first begin executing. GlideIns then fork processes to perform the alloted work, and all system calls from these processes are redirected back to the originating machine. In this way, jobs in Condor-G function similarly to jobs in the original Condor system.

The GlideIn functionality of Condor-G closely mirrors that of placeholders. Both provide the late binding of job to execution entity, an idea that allows for good load balancing. Both also provide the notion of a metaqueue (although Condor-G does not refer to it as such) in which waiting jobs are held until they are assigned to requesting execution entities. However, placeholders differ from GlideIns with respect to the execution model. GlideIns execute Condor-like jobs that redirect system calls back to the originating machine. This provides the illusion that the originating machine has more computational resources (more processors) than it really does. However, in situations that involve a large number of system calls (such as when a large number of jobs are concurrently executed remotely), or when system calls require a large overhead (such as I/O-intensive operations), the overhead of such redirection may be considerable. Placeholders do not reroute system calls and are capable of executing experiments such as that shown in Section 5.3, which involved overheads that would bring any contemporary submitting computer to its knees if used with Condor-G.

2.5 Parameter Space Systems

Parameter space systems are designed to handle situations in which a user wishes to execute the same application multiple times while varying certain execution parameters. For example, the experiment presented in Section 5.3.1 deals with a parameter space study in which the potential energy between two molecules was examined by varying the position of one of the molecules in a three-dimensional grid. Typically, a user wishes to observe the effect that varying a parameter has on the corresponding output so that the impact of the parameter on the overall model can be better understood. Many scientific studies are organized as parameter space studies (including that described in Section 5.3), and so a supportive environment is a valuable tool.

2.5.1 Nimrod

The goal of the Nimrod system [2] is to support parameter space studies common to many scientific computations. Scientific simulations are often performed repeatedly through a range of parameter values. For example, a weather simulation may vary the geographic scale of a weather model from one to one hundred kilometers. Each value in between the end points represents a single run of the simulation.

Nimrod explicitly supports these types of simulation studies. It handles user input, parameter generation, job distribution, and output collection. Users interact with a GUI to provide application information and parameter ranges. The Nimrod system then takes over and executes the application for each parameter value in the specified range. Because each job is independent of the others, Nimrod is able to utilize multiple machines in parallel. The user is able to monitor the progress of the computations, as well as the output that is being created. Essentially, Nimrod is a batch scheduling system similar in form to Condor, but specialized for parameter space studies.

Nimrod is restricted to parameter space studies in which a user wishes to vary parameters over multiple values. TrellisWeb has no such restriction, and instead allows the user to textually specify jobs to be performed. Although this is somewhat more difficult than in Nimrod, it is more general. As a result, TrellisWeb can handle computational studies that do not fit well into the realm of a parameter space system such as Nimrod, along with those that do.

2.5.2 Nimrod/G

Nimrod/G [1] builds upon the original Nimrod by incorporating Globus toolkit components to provide a more advanced parameter space system. Nimrod/G improves upon Nimrod in two major ways: 1) use of the Globus toolkit for resource access; and 2) incorporation of deadlines into scheduling decisions. Nimrod/G is intended to appeal to a larger number of users by accessing grid-enabled resources and by allowing users to specify deadlines for their studies.

The switch from customized resource access systems to Globus infrastructure marks a trend in many metacomputing systems. The original Nimrod had to address each type of system in a customized way, and incorporation of new systems required additions to the Nimrod code. By using Globus for resource access instead of custom code, the responsibility for interfacing with incorporated systems is passed from Nimrod developers onto the Globus toolkit. Also, incorporating existing grid-enabled resources is much easier for Nimrod/G.

Unlike Nimrod/G, placeholder scheduling and TrellisWeb do not yet make use of Globus infrastructure and do not provide deadline-based scheduling. Instead, placeholder scheduling uses existing infrastructure (SSH) and attempts to minimize the makespan for a set of jobs. Attempting to meet a user-specified deadline is largely irrelevant if the makespan is already minimized. Most often, a user is interested in minimizing the entire makespan rather than trying to meet a specific deadline (which may or may not be possible to meet).

2.6 Portals

The main idea behind portals is to make metacomputers more accessible to a wider audience. Metacomputers usually provide some form of command line utilities, which are fine for those who are familiar with command line tools (such as computer scientists). However, an important observation is that metacomputers are targeted at users other than those familiar with command line tools. Should such users be required to interact with a shell to use tools intended to make their life easier? Portals bridge the gap between a user's familiar environment (the web) and the command line interface to metacomputing tools.

2.6.1 GridPort

GridPort [33] is a portal toolkit designed to ease access to Globus grids. Common actions, such as logging in and submitting jobs, can be performed via a command line interface with Globus toolkit components. GridPort allows a user to interact with Globus-enabled resources in the familiar environment of a web browser. Additionally, some aspects of using a Globus grid, such as credential management, can be awkward for users unconcerned with the intricacies of grid security. GridPort manages credentials on behalf of the user, and provides those credentials to the various Globus components as necessary.

GridPort is targeted primarily at Globus grids, just as TrellisWeb is targeted primarily at placeholder scheduling. Both projects acknowledge that command line tools alone are difficult to use, and so develop a more user-friendly environment. TrellisWeb goes a bit further by providing additional benefits such as fault tolerance. Also, placeholder scheduling and TrellisWeb were developed in tandem, which allows for better integration of functionality.

2.6.2 Legion Grid Portal

The Legion Grid Portal [23] is similar in function to GridPort, except for Legion rather than Globus. In addition to making access to Legion metacomputers easier, it also simplifies installation of Legion command line tools. The tools need only be installed in a central location (i.e., the web server), and may be used by any number of users. So, not only can users fill in form information instead of using command line tools, they may not need to install any local software at all. TrellisWeb is similar in that no software must be installed locally for a user to make use of TrellisWeb (except for a web browser, of course).

2.6.3 PUNCH

The Purdue University Network Computing Hubs (PUNCH) system allows users to access and run software tools via the web [15]. PUNCH is based on a three-level system of units: the client unit, the management unit, and the execution unit. Users interact with the client unit via a network desktop interface (through a web browser), and the client acts on behalf of the user. Management units manage interactions between clients and execution units, and execution units are responsible for interacting with the actual resources at a site (e.g., Condor pool). PUNCH is primarily interested in providing easy access to software tools present at one or more sites.

Unlike the other two portals discussed above, PUNCH does not interface with existing metacomputing software. Instead, PUNCH must directly interact with resources, and must hide the complexity of such interactions from the users. As a result, PUNCH software is more complex than simply a front-end for metacomputing software.

Placeholder scheduling and TrellisWeb differ from PUNCH mainly in that placeholder scheduling provides the notion of an overlay metacomputer. While PUNCH may abstract the location of particular tools from users, no attempt is made to schedule work across multiple resources. Also, the PUNCH system multiplexes logical user accounts (PUNCH users) into a single site account. This places the burden of user management on PUNCH rather than the underlying system, a situation that is avoided in TrellisWeb.

2.7 Self-Scheduling

Self-scheduling [27], is a mechanism for scheduling and executing tasks within a parallel application. Self-scheduling is based on the notion of a *local workpile* in which processors each have a private workpile and decide independently when to load balance with other processors. This is in contrast to the more-familiar *global workpile* in which each processor accesses the same workpile when work is required.

Self-scheduling is related to placeholder scheduling in that the processing elements dynamically obtain work to perform. In self-scheduling, the work is continually redistributed amongst the processors. In placeholder scheduling, the placeholders always request additional work when required. Both schemes focus on load balancing through the use of dynamic scheduling of jobs, and neither require an external entity to make scheduling decisions.

Placeholders schedule at the application level instead of within a parallel application. Because of this, placeholder scheduling more closely resembles a global workpile with jobs centralized at the metaqueue. But, because placeholders schedule jobs rather than work within jobs, the contention for the global workpile is significantly reduced from conditions anticipated with self-scheduling. The important common aspect of self-scheduling and placeholder scheduling is that both dynamically load balance work across their respective processing entities.

2.8 Concluding Remarks

With the increasing popularity of metacomputing and metacomputing-like environments, a large number of projects have been developed that address some of the fundamental problems behind metacomputing. Some projects, such as Globus and Legion, envision metacomputers being built out of many diverse resources and are therefore designed to be as comprehensive as possible. EveryWare takes this goal ever further by allowing for the creation of metacomputers out of other metacomputers. Some of the issues addressed by these systems were traditionally the domain of the batch scheduler, and some of the batch schedulers have been developed for metacomputing uses. But ultimately, a metacomputer is about usability, which gives rise to parameter space systems such as Nimrod and Nimrod/G, and Grid portals, such as GridPort and the Legion Grid Portal.

Placeholder scheduling and TrellisWeb were created to provide an alternative to the systems described in this chapter. None of the systems described here (including placeholder scheduling) are amicable to all users and all administrators. If such an ideal system existed, it would be in widespread use due to the demand for metacomputing resources. Placeholder scheduling and TrellisWeb contain many compromises in design decisions, but try to provide similar benefits to other systems while at the same time avoiding some of the less-appealing aspects.

Chapter 3

Placeholder Scheduling

Scheduling is an important component of any metacomputing system. Because metacomputers can cross many administrative domains, the mechanisms behind scheduling can be complex. Even systems that are constrained to a single or small number of administrative domains can have sophisticated scheduling policies [6].

At the core of any scheduling system is the desire to provide increased performance and control over the available resources. Performance may be measured in a number of ways, including throughput (the number of jobs processed by the system in a given time frame), utilization (how much of a resource was in use over a given time frame), response time (how long it took an application to complete), load balancing (a measure of how much work a subsystem performed in relation to how much others performed), and makespan (the total time required to complete a workload), among many others. In general, throughput, utilization, and load balancing should be maximized, whereas response time and makespan should be minimized. It is usually the responsibility of the system developer or administrator to choose the metric(s) of interest.

Placeholder scheduling is a mechanism for creating a metacomputing environment, and a convenient interface for it is described in Chapter 4. Placeholders deal with *how* the actions of a metacomputer are performed. Issues such as remote machine access, job submission, assignment of jobs to various sites, and ensuring a balanced load are all primary aspects of placeholder scheduling. However, placeholder scheduling (as described here) is simple in terms of the policy that is implemented. Placeholders are only used for basic First-Come-First-Served (FCFS) scheduling in which jobs may begin execution in only one possible order. Other work [24] has examined the effects of modifying the scheduling policy to respect inter-job dependencies.

One of the primary features of placeholder scheduling is its ability to function in the absence of a ubiquitous software infrastructure (beyond that of SSH). In order to use placeholder scheduling on a system, a user merely requires an account and a means of accessing that account. Placeholder scheduling is flexible enough to work with only these requirements, and this makes it a useful alternative to some other, better-established metacomputing systems (such as Globus).

Parameter	**Batch Scheduler Binding Time**	**Placeholder Binding Time**
Job Name	Submission	Submission
Shell	Submission	Submission
Machine	Submission	Submission
Queue	Submission	Submission
Maximum Run Time	Submission	Submission
Mail Options	Submission	Submission
Executable Name	Submission	**Execution**
Arguments	Submission	**Execution**

Table 3.1: Batch Scheduling vs. Placeholder Scheduling

3.1 Overview

A *placeholder* is defined as a unit of potential work. A placeholder is capable of performing a wide range of tasks, or even no task at all. Placeholders operate as a group, with each group responsible for a set of jobs waiting to be performed. Placeholders in a group work as a team to complete all waiting jobs stored in a job repository. Each such group forms an overlay metacomputer for a particular set of jobs.

Placeholders work based on the observation that only certain information is required when queuing a job. Table 3.1 presents some parameters and the time at which they are required for batch scheduling and placeholder scheduling. Because certain parameters, such as the executable, are not required until the placeholder actually starts, we can delay the binding of job to placeholder until the time at which the placeholder is prepared to execute. We call this the *late binding* of job to placeholder.

Placeholders are able to take advantage of multiple machines by exploiting this late binding property. When a placeholder is eligible to execute on a machine, it contacts a central authority for information about which application to execute. It is the sole responsibility of this authority to allocate jobs to placeholders. Multiple placeholders on multiple machines make contact with the same authority, and hence any one may receive any available job. Users select how many placeholders should be assigned to each machine. In the event that no more jobs are available, the placeholders terminate with minimal resource consumption.

Consider Figure 3.1 as a small example. Here, there are three queues with one placeholder each. Each of the placeholders must pull jobs from the metaqueue. The thickness of the lines indicates the frequency of pulling jobs. In this case, placeholder PH2 demands jobs most frequently, and PH1 demands least frequently. As a result, PH2 will execute a larger proportion of jobs in the metaqueue, and PH1 will execute less. Exactly which jobs are allocated to which placeholder is largely unimportant, so long as the load is balanced over all placeholders and the overall makespan is reduced.

Late binding of job to placeholder also affords performance benefits. A placeholder that requests

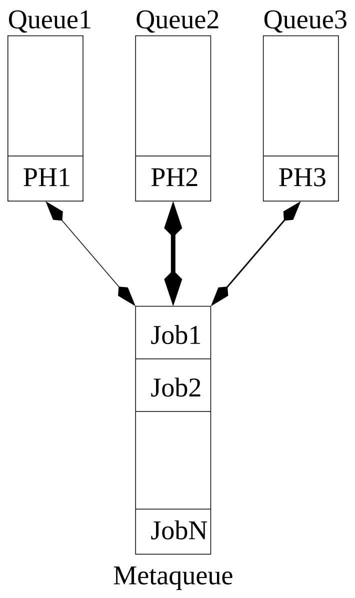


Figure 3.1: Placeholder Scheduling Example

a job is a placeholder that is prepared and fully capable of executing at that exact moment in time. As a result, the assigned job has waited a low amount of time to begin executing. Hence, it has a low queuing time. The throughput of the entire set of jobs is maximized by repeatedly binding the next available job to a placeholder prepared to execute. By binding only to placeholders prepared to execute, “slow” systems (systems that have a relatively low rate of execution due to high utilization or slower processors) are avoided. Additionally, load balancing is maximized because no capable placeholder remains idle while there are available jobs.

Placeholder scheduling is purposely designed as a *pull model* rather than a *push model*. In a push model, a central authority actively identifies available resources (e.g., idle processors) and assigns jobs to them. A push model is a natural way to implement a scheduling system, as the central authority has knowledge of all jobs. A pull model, on the other hand, has the resources (e.g., placeholders) actively contact a passive central authority when ready to execute. The benefits of a pull model are:

1. The problems of admission control of placeholders, placing jobs, and fault tolerance are orthogonal. The local batch schedulers are responsible for deciding exactly when placeholders will be allocated resources. A user selects which sites will be used with placeholder scheduling (and how many placeholders should be started on each machine), but the batch scheduler has the final say. Orthogonal to this process is the issue of job placement. Placeholders pull jobs when they are prepared to execute (i.e., permitted to execute by the underlying scheduler). Furthermore, detecting and correcting faults are orthogonal to these issues because

faults can be isolated to an individual placeholder and may be dealt with independently of other placeholders and their activities. Restarting simply involves recycling the incomplete job and submitting a new placeholder. This orthogonality affords the pull model a great deal of flexibility that may not otherwise be available.

2. Jobs are self scheduling in the pull model, which requires less overhead and results in easier load balancing. There is less overhead because placeholders are only considered when they require work. A push model would have to consider a number of placeholders when work is available so that jobs can be pushed to them. By having placeholders pull, a central authority only has to respond when contacted. Likewise, load balancing is easier because placeholders pull at a rate that is equivalent to their computational abilities. A placeholder will not become unbalanced if it obtains work as fast as it is able to compute it.

3. Placeholders contain more state information in a pull model. Placeholder scheduling can work in situations in which the central authority has no knowledge of placeholder state. The placeholder simply stores this information and provides it when pulling jobs. As a result, placeholders are capable of making "smart" decisions based on the state information they contain. Such decisions may include dynamically increasing or decreasing the number of placeholders present at a site. Push models require more state information to reside in the central authority and would therefore have "dumb" placeholders.

Placeholder scheduling does not address the issue of executable or data staging. Other work on data grids (e.g., the Trellis Filesystem [29]) explores some of the problems of accessing data files. Although executable and data file staging are important issues, underlying systems may already support this task. Instead, the onus for executable and data staging is placed on the user (with the assistance of the TrellisWeb system, discussed in Chapter 4), thereby allowing placeholder scheduling to focus purely on scheduling jobs.

Figure 3.2 presents a high-level, graphical overview of placeholder scheduling. Sites 1 through n represent the overlay metacomputer for this placeholder group. Each site of the overlay metacomputer has one or more placeholders present. For example, Site 1 has three placeholders (one in each batch scheduler queue) that have reached the front of the queue and have begun executing. Site n, on the other hand, is not batch scheduled (zero-infrastructure) and has only one special zero-infrastructure placeholder. Each of the placeholders make contact with a central authority (called the command-line server, and discussed in Section 3.4) via SSH to obtain work (late binding). The command-line server is responsible for selecting and returning executable information to each of the requesting placeholders. Other placeholder and non-placeholder jobs may be present in the queues, but are omitted for clarity.

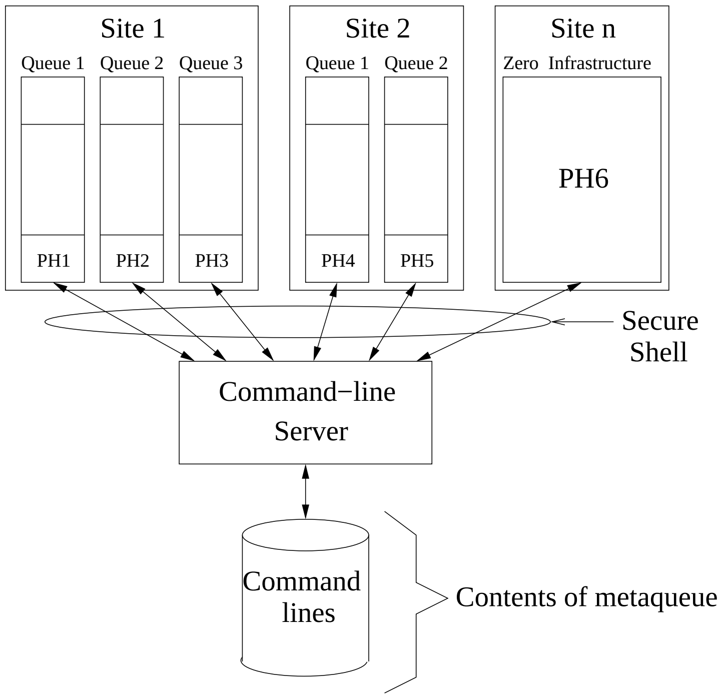


Figure 3.2: Placeholder Scheduling Overview

3.2 Advantages of Placeholder Scheduling

Placeholder scheduling is certainly not the only method for running jobs across multiple systems. Placeholder scheduling, as a mechanism for metacomputing, is advantageous for several reasons:

1. **Minimal Infrastructure Requirements**: Placeholders are often implemented as batch scheduler jobs for batch scheduled sites, or as shell scripts for zero-infrastructure sites. This means that no additional software is required for placeholders to execute on these machines. Furthermore, access to the sites requires only a login account so that the user (or proxy on the user's behalf) may submit or start the placeholder. Some other metacomputing systems require a substantial amount of software infrastructure to be in place before a user may run applications.

2. **Resource Heterogeneity**: Placeholders may be implemented in any desired language, provided they exhibit correct placeholder behaviour (further discussed in Section 3.3). The easiest way to implement this behaviour is via a job or shell script, but compiled programs may also suffice. Although placeholders are targeted mainly at high-performance batch scheduled systems, the flexibility of placeholders allows any system, batch scheduled or not, to participate in placeholder scheduling.

3. **Loose Coupling of Participant Sites**: Most systems that provide a metacomputing environment require some degree of integration of participant sites – that is, the sites must agree to certain software and policies governing the use of that software (e.g., Globus). Although resolving these issues may be trivial within a single administrative domain, larger metacomputers will inevitably involve the crossing of one or more administrative boundaries. Placeholder scheduling minimizes the integration of participant sites by allowing each site to set its own policies. Placeholders are executed as user-level processes, and are therefore subject to local policy. Moreover, different users may specify different overlay metacomputers to use during scheduling, a situation that would require union of all possible metacomputers to be integrated into a metacomputing system.

4. **Cooperation with Underlying Schedulers**: Although a metacomputing environment conveys a large number of benefits to users, one must realize that some users are content to simply use resources at a single site and will therefore not require a metacomputing environment. If a metacomputing environment were to replace the underlying access system, such users would have no choice but to use the metacomputing environment. Placeholder scheduling does not replace underlying scheduling systems, but rather layers on top of them and makes use of their job submission services. Placeholders are also able to operate in the absence of any scheduling system without requiring full control of the computer. By cooperating with them, much of the scheduling work can be offloaded to the local schedulers.

5. **Improved Performance**: Placeholders are more than tools used to access multiple sites. They are able to load balance a set of jobs at high throughput and with low queuing times across a number of sites. This is possible because of the late binding of job to placeholder. Eligible placeholders contact a central job repository for the next available job. Thus, jobs are served only to those placeholders currently capable of executing, and “slow” placeholders are avoided. Also, a greater proportion of executables will be served to placeholders that complete their work quickly as opposed to those that work more slowly, which ensures a balanced load.

Placeholder scheduling is largely a mechanism for metacomputing. In terms of policy, placeholders are simple. However, the scheduling policy implemented by placeholders is not strictly limited to simple schemes. One interesting property of placeholders is the retrieval of jobs from a central repository. In the simplest policy, the jobs are served in the order they were entered into the repository. In previous work [24], jobs were served according to pre-determined dependencies. These dependencies were strict in that subsequent jobs required the results of previous ones before they could begin. Dependency-based job ordering is just one possible complex policy that placeholder scheduling may be used to implement.

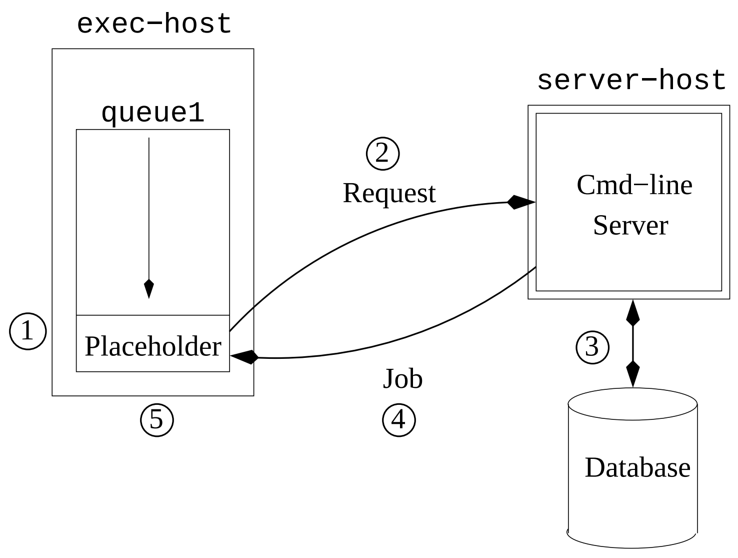


Figure 3.3: Steps in Placeholder Execution

3.3 Placeholders in Action

In order for placeholder scheduling to work correctly, each placeholder must adhere to a protocol that regulates the binding of jobs to placeholders. In addition to placeholders, there are several other entities that are part of the binding process. The overall binding process is shown in Figure 3.3.

The steps in the executable binding process are as follows:

1. A placeholder has reached the front of `queue1` on `exec-host` and has begun executing. In the event that there is no batch scheduler, the placeholder begins executing as soon as the host operating system allocates processor time to the process. The placeholder is currently capable of executing many tasks, and is awaiting binding to a specific task.

2. The placeholder requests an available job from the central repository of jobs (via the command-line server, discussed in Section 3.4) on `server-host`. The placeholder is now obligated to perform any work that the command-line server returns, in addition to any other static work required by the placeholder on this particular system (such as authentication, for example).

3. The command-line server consults a persistent database of jobs. Although the jobs are stored according to some ordering within the database, the command-line server is free to select any available job based on constraints. Therefore, the command-line server can impose any arbitrary ordering on the jobs in the database.

4. A selected job is returned to the waiting placeholder. The returned job is removed from the database so that no other placeholder may obtain a duplicate job. The placeholder does not perform any computation while waiting for the job, and is essentially blocked until one is

returned. Therefore, it is imperative that the command-line server return a job as expediently as possible.

5. The placeholder, with job in hand, is permitted to continue execution, eventually performing the work specified in the job. At this time, the placeholder is fully bound to the job, and will not contact the command-line server until it requires work once again.

Once a placeholder completes its assigned task, it renews itself according to the underlying scheduling system. In the presence of a local batch scheduler, the placeholder resubmits itself to the same queue. This policy allows a one-to-one correspondence between placeholder job and local job. In addition, the resubmission allows jobs from other users an opportunity to execute. In the absence of a local batch scheduler, the placeholder is free to repeatedly request jobs until none remain.

3.4 Command-Line Server

The command-line server is the authority responsible for binding jobs to placeholders. Each placeholder that is prepared to execute must contact the command-line server to obtain the information required to continue execution. The assignment policy of jobs to placeholders is implemented wholly within the logic of the command-line server. As a result, the command-line server implements the global policy of placeholder scheduling. The command-line server may also choose to deny the placeholder a job (e.g., when no more jobs remain in the metaqueue), effectively halting the execution of that placeholder.

The jobs for the command-line server are stored in some persistent form, such as in a database or filesystem. As more than one group of placeholders may be executing at any given time, the jobs must be keyed according to the group they belong to. The order in which the jobs are stored in the database is unimportant, as the command-line server is free to select from any available job for the placeholder group in the database.

The command-line server is also responsible for ensuring that each job is executed by only one placeholder. Violation of this constraint may have adverse effects, such as corrupted data files. Typically, requests made by placeholders are serialized and dealt with in some sequential order. The ordering imposed by the command-line server may be as simple as FCFS, but may be more complex, servicing requests in some other order (for example, using a priority scheme). This at-most-once execution of jobs may not hold if a fault occurs at one or more of the sites (see Section 4.5 for more details on fault tolerance).

A command-line server may be implemented in any number of ways. A simple method is to return the jobs from the database in the order they are entered (i.e., FCFS). This policy is easy to implement and also exhibits predictable behaviour. Such a simple policy places the burden of prioritizing the jobs onto the user. Jobs entered in the database earlier are implicitly of higher priority that those entered later. To prioritize jobs, the user simply adjusts the ordering of the jobs. This simple policy is the one used in the implementation described in Section 3.5.5.

A command-line server may also be priority-based. Users may attach priorities to jobs entered into the database such that higher priority jobs are executed before lower-priority jobs. Although this scheme is implicit with the FCFS policy described above, the main advantage of explicit priority is that the user does not need to order jobs to specify priority. Newly added jobs may therefore be served prior to some or all of the jobs previously in the database. Care must be taken by the user not to starve existing jobs by repeatedly adding higher-priority jobs.

Another command-line server (which was implemented and used in [24]) serves jobs based on directed acyclic graph (DAG) dependencies. The edges of the DAG represent dependencies, and the nodes represent jobs. In order for a job to be eligible to execute (i.e., served by the command-line server), all dependency edges leading from that node must be satisfied (i.e., the prerequisite jobs must have been completed). A dependency-based command-line server allows workflows to be easily specified by the user. The parallelism inherent in the workflow can be exploited without the need to manually break up the workflow into individual parts or stages. The full details of dependency based job scheduling are beyond the scope of this thesis and are discussed elsewhere [12].

The command-line server is a critical component of placeholder scheduling. All of the logic required to obtain a unit of work is concentrated in the command-line server. The server implements the policy of obtaining work, and this policy may be as simple or complex as desired. The command-line server also allows all placeholders to contact a single entity when in need of work, rather than requiring the placeholders to make decisions for themselves.

3.5 Implementation of Placeholder Scheduling

Placeholders, and the associated command-line server, may be implemented in a number of different ways so long as they adhere to the functionality described in Section 3.3. Both placeholders and the command-line server are free to perform tasks in addition to those required to carry out placeholder scheduling, especially those tasks that allow the components to better handle heterogeneous environments (e.g., special commands for certain operating systems) and faults (e.g., crashes). All components may be implemented in any desired language, and they may be mixed and matched for greater flexibility. Without this flexibility in implementation, placeholder scheduling would not be as effective in handling heterogeneity.

The implementations described here are intended as reference implementations only. The implementations of placeholders used in this thesis are written as shell scripts because the majority of batch scheduling systems work on job scripts. These job scripts must be written as shell scripts, so placeholders naturally fit as shell scripts. The portability of job scripts between different machines with identical batch scheduling systems is also beneficial. The command-line server does not require the same degree of portability because it must only run on a single machine, but is best written as a script so that it can be easily migrated to another machine.

3.5.1 Placeholders

The notion of a placeholder is intended to be as simple as possible and, as such, placeholders can be implemented on a variety of systems. Placeholders can be implemented to exploit the underlying batch scheduling system, thereby allowing other users and placeholders an opportunity to execute. An underlying scheduler also provides the benefits of local job scheduling, submission, monitoring, and management. These benefits are as equally applicable to placeholders submitted via the batch scheduling interface as they are to normal jobs. An underlying batch scheduler is therefore the preferred implementation method, when possible.

Unfortunately, underlying batch scheduling systems do not exist at all sites. In the event of, say, a dedicated system, there may be little need for a batch scheduler. Therefore, placeholder scheduling should not require the existence of a batch scheduler, which would make these systems inaccessible. As a result, placeholders may be implemented as *zero-infrastructure* placeholders. Such an implementation makes no use of an underlying batch scheduler, and instead accesses the machine directly by creating processes itself.

Each of the implementations are presented next. Placeholders are written as abstract shell scripts (i.e., some of the details are omitted for clarity), but are similar in form to a minimal placeholder. As stated earlier, placeholders may include code to perform additional tasks. Likewise, the command-line server is presented as pseudocode to illustrate its intended functionality.

3.5.2 Batch Scheduled Placeholders

Figure 3.4 presents an example placeholder for PBS. At only 30 lines in length, the placeholder is compact. The preamble of the script (lines 1–13) contains PBS-specific directives. These directives control parameters such as the job name (line 7), the number of processors required (line 9) and the time limit of the job (line 10), among others, and are fixed at job submission time. Other scheduling systems, such as Sun Grid Engine, have similar parameters that are specified differently.

When a placeholder begins executing, its first task is to contact the command-line server. The remote program `getcmdline` is used to contact the command-line server. The placeholder connects to the command-line server machine (`$CLS-MACHINE`, line 21), and executes `getcmdline`, passing the contents of `$ID-STR` as arguments. `$ID-STR` is intended to contain identifying or state information about the placeholder (for identification and authentication), but may contain nothing at all if no such information is required.

The command-line server may wish to inform this placeholder to discontinue execution. This is done by returning a non-zero value from `getcmdline`, and is checked by the placeholder on lines 22–24. The placeholder then executes the job returned by the command-line server (line 27).

```
1  #!/bin/sh
2
3  ## Simple placeholder for PBS.
4
5  ## PBS-specific options.
6  #PBS -S /bin/sh
7  #PBS -N Placeholder
8  #PBS -q dque
9  #PBS -l ncpus=1
10 #PBS -l walltime=02:00:00
11 #PBS -j oe
12 #PBS -M pinchak@cs.ualberta.ca
13 #PBS -m ae
14
15 ## Environment variables:
16 ##   CLS-MACHINE - points to the command-line server’s host.
17 ##   CLS-DIR - remote directory in which the command-line server
18 ##             is located.
19 ##   ID-STR - information to pass to the command-line server.
20 ## Note the backquote, which executes the quoted command.
21 JOB=‘/usr/local/bin/ssh $CLS-MACHINE "$CLS-DIR/getcmdline $ID-STR"‘
22 if [ $? -ne 0 ]; then
23     exit 111
24 fi
25
26 ## Execute the command from the command-line server.
27 eval $JOB
28
29 ## Resubmit the placeholder to "play nice" with the scheduler.
30 qsub placeholder.pbs
```

Figure 3.4: Simple PBS Placeholder

Finally, the placeholder resubmits itself to the batch scheduling system using the provided interface (line 30). This is done to allow other batch scheduler jobs an opportunity to execute. It also allows an arbitrary number of low-limit placeholder invocations to execute, as PBS may kill any job exceeding the specified limits.

3.5.3 Zero-Infrastructure Placeholders

An example zero-infrastructure placeholder is presented in Figure 3.5. The zero-infrastructure implementation of a placeholder shares many similarities with a batch scheduled implementation. The preamble (lines 6–7) simply redirects output to a file instead of to the terminal (as a batch scheduler would do). The remainder of the placeholder is no different from the batch scheduled implementation, save for the use of the `zinfqsub` program. `zinfqsub` (shown in Figure 3.6), is simply a script designed to mimic the non-blocking job submission semantics of many batch scheduling systems.

3.5.4 Self-Regulating Placeholders

A PBS placeholder with the ability to regulate the total number of placeholders on the system is presented in Figure 3.7. Such a placeholder may be advantageous in situations in which queue conditions are highly variable. When a queue is determined to be “fast”, a self-regulating placeholder could submit additional placeholders to take advantage of the current conditions. Likewise, the number of placeholders can be reduced if conditions are such that placeholders must wait in the queue

```
#!/bin/sh

## Simple zero-infrastructure placeholder.

## Redirect output.
exec 1>Placeholder.o$$
exec 2>&1

## Environment variables:
##   CLS-MACHINE - points to the command-line server’s host.
##   CLS-DIR - remote directory in which the command-line server
##             is located.
##   ID-STR - information to pass to the command-line server.
## Note the backquote, which executes the quoted command.
JOB=‘/usr/local/bin/ssh $CLS-MACHINE "$CLS-DIR/getcmdline $ID-STR"‘
if [ $? -ne 0 ]; then
    exit 111
fi

## Execute the command from the command-line server.
eval $JOB

## Resubmit using a special script.
zinfqsub placeholder.zinf
```

Figure 3.5: Simple Zero-infrastructure Placeholder

```
#!/bin/sh

/bin/sh $@ &
echo $!
```

Figure 3.6: Non-blocking Zero-infrastructure Submission Script

for a long period of time.

A self-regulating placeholder keeps track of the time it spends in the queue by recording the current time in a file (line 54) immediately before exiting. The subsequent placeholder (submitted on line 52) consults this file when it begins executing (line 25), and calculates the difference between the present time and when the file was written (lines 27–31). A local program (`decide`) is used to determine what action the placeholder should take (line 34). The possible responses (for this placeholder, at least) are `increase`, `decrease`, or `maintain`. If the decision is to increase placeholders, a new placeholder is started immediately (i.e., before the job is performed) (lines 36–39) and the current placeholder resubmits itself before exiting (line 52). If the decision is to decrease the number of placeholders, the current placeholder simply does not resubmit itself on exit (lines 45–47). And, if the decision is to maintain the current number of placeholders, the current placeholder simply resubmits itself as usual (line 52). Thus, the number of placeholders present at a site is incrementally adjusted by the self-regulating placeholders.

3.5.5 The Command-Line Server

Although the command-line server has been presented as a *server* (with the implication of a client-server model), our reference implementation is instead a script remotely executed by a placeholder. Because the command-line server script can be complex, a pseudocode version is presented in Fig-

```
#!/bin/sh
## Self-regulating PBS placeholder.

## PBS-specific options.
#PBS -S /bin/sh
#PBS -N Placeholder
#PBS -q dque
#PBS -l ncpus=1
#PBS -l walltime=02:00:00
#PBS -j oe
#PBS -M pinchak@cs.ualberta.ca
#PBS -m ae

## Environment variables:
##   CLS-MACHINE - points to the command-line server’s host.
##   CLS-DIR - remote directory in which the command-line server is located.
##   ID-STR - information to pass to the command-line server.
## Note the backquote, which executes the quoted command.
JOB=‘ssh $CLS-MACHINE "$CLS-DIR/getcmdline $ID-STR"‘

if [ $? -ne 0 ]; then
    exit 111
fi

STARTTIME=‘cat $HOME/MQ/$PBS_JOBID‘
NOWTIME=‘$HOME/bin/mytime‘
if [ -n "$STARTTIME" ] ; then
    let DIFF=NOWTIME-STARTTIME
else
    DIFF=-1
fi

## Decide if we should increase, decrease, or maintain placeholders in the queue.
WHATTODO=‘$HOME/decide $DIFF‘

if [ $WHATTODO = ’increase’ ]; then
    NEWJOBID=‘qsub placeholder.pbs‘
    $HOME/bin/mytime > $HOME/MQ/$NEWJOBID
fi

## Execute the command from the command-line server.
eval $JOB

## Leave if ’reduce’.
if [ $WHATTODO = ’reduce’ ] ; then
    exit 0
fi

/bin/rm -f $HOME/MQ/$PBS_JOBID

## Recreate ourselves if ’maintain’ or ’increase’.
NEWJOBID=‘qsub placeholder.pbs‘

$HOME/bin/mytime > $HOME/MQ/$NEWJOBID
```

Figure 3.7: Generic PBS Placeholder

```
1  ## Assume that ID_STR is passed in as an argument.
2  if not verify(ID_STR) then
3      return failure_value
4
5  lock database
6
7  ## Select the next available job based on the contents of ID_STR.
8  next_job := select-next-job(ID_STR)
9
10 ## Check if anything was available.
11 if next_job = "" then
12     unlock database
13     return stop_value
14
15 unlock database
16
17 ## Print out the job information (so it can be captured).
18 print next_job
19 ## The return value indicates a success or failure.
20 return success_value
```

Figure 3.8: Pseudocode Command-line Server Script

ure 3.8.

Execution of the command-line server is as follows. First, the command-line server must verify that a legitimate placeholder is attempting to obtain a job (line 2–3). This represents a minimal level of security in that someone trying to spoof a placeholder must know this information *a priori*. Next, the database must be locked to ensure mutual exclusion (line 5). This is important because multiple placeholders may invoke the command-line server script concurrently, and we must ensure that each invocation returns a previously unassigned job. Once the database is locked and mutual exclusion is ensured, the script is free to select the next job (the policy of which is implemented in the `select-next-job()` function) from the database (line 8). In the event that there are no remaining jobs, the placeholder must be notified to halt execution because all work has been performed for the placeholder group (lines 11–13). Finally, should a job be available, the job information is printed (so that it may be captured by the placeholder) and success is reported.

A considerable amount of information has been omitted from Figure 3.8. For example, locking the database (line 5) may be as simple as requesting that the underlying database system lock the database, or as complex as manually implementing mutual exclusion. Likewise, the selection of the exact job to return is abstracted by the `select-next-job()` function. Although a concrete, working command-line server has been developed for the experiments in Chapter 5, the full program is much too large to be adequately described here.

3.6 Concluding Remarks

Placeholder scheduling provides a metacomputing environment with performance benefits for relatively low complexity. Through the mechanism of placeholders, the benefits are provided with minimal infrastructure and integration of participating sites. Placeholders are able to handle a great deal of resource heterogeneity (batch scheduled and zero-infrastructure systems) and cooperate with underlying schedulers to avoid interfering with existing operations. The largest benefit, perhaps, is that placeholder scheduling requires nothing more than what can be easily allocated to a normal user account. Therefore, placeholder scheduling is appealing from both the user's (improved performance) and administrator's (no special treatment) point of view.

Chapter 4

TrellisWeb

Placeholder scheduling is a useful mechanism for taking advantage of computational resources at multiple sites. The simplicity of placeholders makes them an attractive alternative to other existing scheduling systems. However, using placeholder scheduling can be somewhat difficult. Placeholders must be created for each system (often by hand), and must somehow be distributed to each location. Additionally, the user must somehow start, stop, and monitor the progress of the placeholders as they do their work. Any one of these tasks becomes a major chore when the number of sites participating in an overlay metacomputer becomes large.

TrellisWeb was designed to address these user-level complications. By offering a single web-based interface to placeholder scheduling, TrellisWeb allows non-trivial metacomputers to be composed and managed by non-expert users. TrellisWeb also provides a certain level of fault tolerance for placeholder scheduling. Without these simplifying features, placeholder scheduling would be a tedious task.

The target audience of placeholder scheduling is those members of the scientific community pursuing computationally-intensive research. The primary focus of such research is not the underlying details of performing the computations, but rather interpreting the results. Because the computations are necessary, and only the results are interesting, it stands to reason that time spent on conducting the computational research is less valuable than time spent on examining and interpreting the results. TrellisWeb attempts to address this issue by requiring as little user intervention as possible so that researchers may spend time in a more productive manner.

4.1 Overview

TrellisWeb is based on the PBSWeb system [20, 21]. PBSWeb was originally designed as a web-based graphical user interface (GUI) to the Portable Batch System. The goal of PBSWeb is to make PBS job submission easier and more intuitive for novice users. Many of the PBS commands are complicated beyond the abilities of the casual user, and the forms-based GUI provided by PBSWeb was purposefully designed to allow for easy selection of the most common options. It is hoped that the ease of use of PBSWeb will encourage scientists in areas other than computing science to develop and use computationally-intensive applications that require high-performance computing resources.

TrellisWeb augments PBSWeb functionality with the ability to create and manage overlay metacomputers. Users can upload applications to multiple computers, create an overlay metacomputer, specify placeholder parameters, load jobs into the per-metacomputer metaqueue, monitor placeholder progress, and start and stop placeholder execution. All actions are performed on behalf of the user via a web browser. Therefore, all actions have the same effect as a user performing them manually.

It was a deliberate decision to use a web-based portal instead of command line tools. Command line tools are familiar to most computer scientists, but not to target users such as scientists in other areas. Due to the great increase in computational aspects of various disciplines, areas other than computer science would benefit from a metacomputing environment. Command line tools for placeholder scheduling would likely become unnecessarily arcane, causing problems even for those familiar with such tools. Instead, command line tools are foregone in favour of the cross-platform support of the more user-friendly web interface.

4.2 Key Features

TrellisWeb provides a number of benefits to users of placeholder scheduling. Briefly, they are:

1. **Placeholder management**: Placeholders can be difficult to use, especially when they are required for a large overlay metacomputer. As a result, placeholder management is the primary goal of TrellisWeb. All aspects of placeholder management are handled by TrellisWeb, including creation and customization of placeholder scripts, uploading of placeholder scripts to destination sites, starting and stopping placeholders, and monitoring the progress of the executing placeholders. TrellisWeb tries to present this information as if the overlay metacomputer were actually a single cohesive system, where possible.

2. **Metaqueue interaction**: Placeholder scheduling is based on the dynamic assignment of jobs to placeholders. As discussed earlier, this information is provided to placeholders via the command-line server. TrellisWeb provides a forms-based method for adding and removing jobs to/from the command-line server database. New jobs may be added by the user in stages or all at once, and can be selectively removed. Metaqueue interaction is provided by TrellisWeb so that the user may avoid using potentially-complex SQL database queries to perform similar actions.

3. **Basic fault tolerance**: As the number of sites participating in an overlay metacomputer increases, so does the likelihood of a fault. Fault tolerance is itself a large research topic, and

so is addressed only in a basic form in TrellisWeb. TrellisWeb records when and where jobs are assigned and is capable of restarting jobs should they become suspect of failure (i.e., they have been in the executing state for an excessive amount of time). Because the fault tolerance is only basic, no attempt is made to ensure once and only once execution of all jobs in a metaqueue when faults occur. Files may be corrupted due to the partial completion and re-execution of jobs. The fault tolerance of TrellisWeb is only intended to catch the majority of common errors so that simple failures can at least be corrected.

In addition to the above features, TrellisWeb records information such as the assignment of jobs to placeholders, the time the placeholder began executing a job, and the time the placeholder finished executing a job. Such information is invaluable for post-mortem analysis of job execution. TrellisWeb stores all placeholder actions in a database for later analysis. The experimental results presented in Section 5.3 use such logging features.

4.3 TrellisWeb Walkthrough

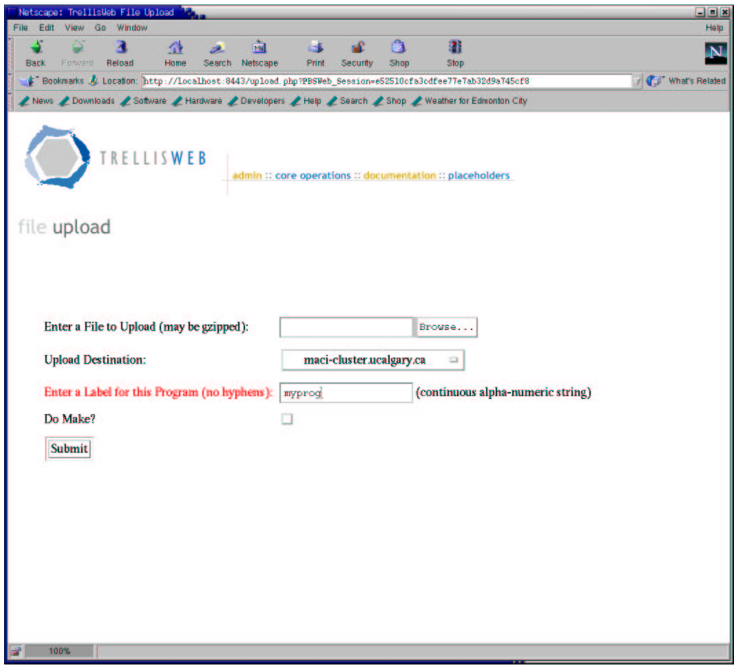
TrellisWeb provides a start-to-finish process to allow users to execute jobs. It is useful to examine this process in detail, as certain design decisions have been made that affect both user interaction and the generality of placeholder scheduling within TrellisWeb. The following sections discuss each step of placeholder scheduling with TrellisWeb.

4.3.1 Application Upload

While it is certainly not the most exciting aspect of metacomputing, ensuring that executables are placed at all required locations is extremely important. Some systems, such as Condor-G, incorporate automatic and transparent executable staging into the system itself. While this is convenient for the user, it increases the complexity of the system because of the need for architecture identification and appropriate executable selection.

Of course, TrellisWeb still provides some support for application upload (see Figure 4.1). Users may select any Unix Tape Archive (TAR) file to upload to any location they have access to in the TrellisWeb system. In this particular example, an application will be uploaded to the University of Calgary MACI Alpha cluster site. TrellisWeb administrators must add new systems to TrellisWeb, but users themselves may select which they wish to access and which account is to be used. Users give a label to the upload and select whether or not `make(1L)` should be invoked on the contents of the archive. The archive is then securely copied and extracted at the destination. Care is taken so that archives uploaded to sites with different operating systems/architectures that share a common filesystem do not interfere with each other.

Once this step has been completed, we can say that the *project* named by the label exists at the destination. The name and location of the project will later affect the composition of the over-


Netscape: TrellisWeb File Upload
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8443/upload.php?PBSWeb_Session=e52510cfa3cdfee77e7ab32d9a745cf8 What's Related
News Downloads Software Hardware Developers Help Search Shop Weather for Edmonton City
TRELLISWEB
admin :: core operations :: documentation :: placeholders
file upload
Browse...
Submit
100%

Enter a File to Upload (may be gzipped):	
Upload Destination:	maci-cluster.ucalgary.ca
Enter a Label for this Program (no hyphens):	myprog (continuous alpha-numeric string)
Do Make?	☐

Figure 4.1: TrellisWeb Application Upload

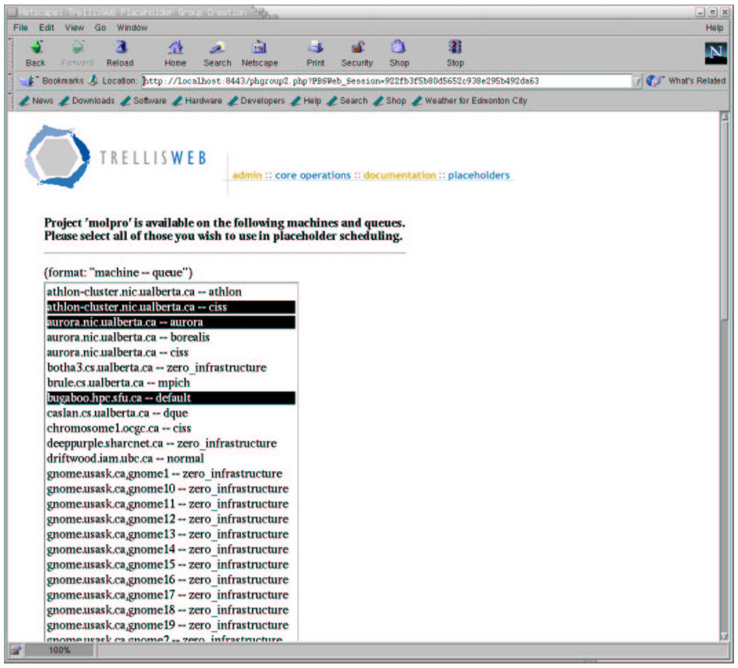
lay metacomputer. This step must be repeated for each site that will compose the desired overlay metacomputer.

4.3.2 Placeholder Group Creation

Once the projects have been uploaded to the various locations, the next step toward placeholder scheduling can be performed. The user must now specify the composition of the overlay metacomputer. In the previous step, projects were assigned a label. TrellisWeb considers identical labels to be similar projects, and hence such projects are candidates for a group. For example, if a user were to upload project "`myprog`" to the University of Calgary, and another, different, project "`myprog`" to the University of Alberta, TrellisWeb would allow the user to placeholder schedule across both sites using the respective "`myprog`", even though the projects may be different. Therefore, users must be mindful of the labels they provide when uploading files.

Users may now select a subset (proper or not) of the sites with a given project to form the overlay metacomputer. An example group creation page for the "`molpro`" project is shown in Figure 4.2. Although an overlay metacomputer (also known as a placeholder group in TrellisWeb) is application-specific, multiple identical overlay metacomputers may be created, and overlay metacomputers may be used for tasks other than those associated with a particular project. Additional parameters (both site independent and site specific) may be specified on the following page (see Figure 4.3).

Upon specification of the additional parameters, placeholder scripts are created and distributed

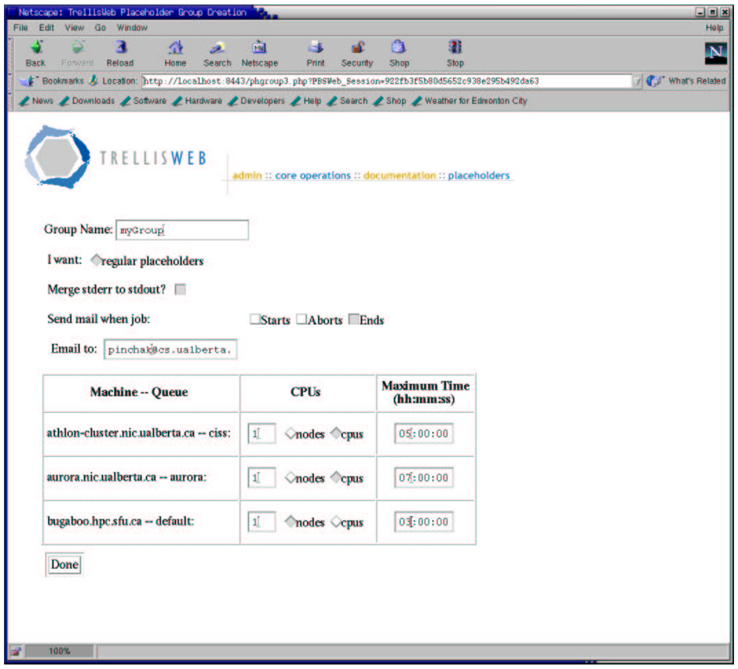

Netscape: TrellisWeb Placeholder Group Creation
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8443/phgroup2.php?PBSWeb_Session=922fb3f5b80d5652c938e295b492da63
What's Related
News Downloads Software Hardware Developers Help Search Shop Weather for Edmonton City
TRELLISWEB
admin :: core operations :: documentation :: placeholders
100%

Project 'molpro' is available on the following machines and queues.
Please select all of those you wish to use in placeholder scheduling.

(format: "machine -- queue")

athlon-cluster.nic.ualberta.ca -- athlon
athlon-cluster.nic.ualberta.ca -- ciss
aurora.nic.ualberta.ca -- aurora
aurora.nic.ualberta.ca -- borealis
aurora.nic.ualberta.ca -- ciss
botha3.cs.ualberta.ca -- zero_infrastructure
brule.cs.ualberta.ca -- mpich
bugaboo.hpc.sfu.ca -- default
caslan.cs.ualberta.ca -- dque
chromosome1.ocgc.ca -- ciss
deeppurple.sharcnet.ca -- zero_infrastructure
driftwood.iam.ubc.ca -- normal
gnome.usask.ca,gnome1 -- zero_infrastructure
gnome.usask.ca,gnome10 -- zero_infrastructure
gnome.usask.ca,gnome11 -- zero_infrastructure
gnome.usask.ca,gnome12 -- zero_infrastructure
gnome.usask.ca,gnome13 -- zero_infrastructure
gnome.usask.ca,gnome14 -- zero_infrastructure
gnome.usask.ca,gnome15 -- zero_infrastructure
gnome.usask.ca,gnome16 -- zero_infrastructure
gnome.usask.ca,gnome17 -- zero_infrastructure
gnome.usask.ca,gnome18 -- zero_infrastructure
gnome.usask.ca,gnome19 -- zero_infrastructure
gnome.usask.ca,gnome2 -- zero_infrastructure

Figure 4.2: Placeholder Group Creation (step one)


Netscape: TrellisWeb Placeholder Group Creation
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8443/phgroup3.php?PBSWeb_Session=922fb3f5b80d5652c938e295b492da63
What's Related
News Downloads Software Hardware Developers Help Search Shop Weather for Edmonton City
TRELLISWEB
admin :: core operations :: documentation :: placeholders
Done
100%

Group Name: myGroup

I want: [✓ regular placeholders]

Merge stderr to stdout? [✓]

Send mail when job: Starts Aborts [✓ Ends]

Email to: pinchak@cs.ualberta.

Machine -- Queue	CPUs	Maximum Time (hh:mm:ss)
athlon-cluster.nic.ualberta.ca -- ciss:	1 nodes [✓ cpus]	05:00:00
aurora.nic.ualberta.ca -- aurora:	1 nodes [✓ cpus]	07:00:00
bugaboo.hpc.sfu.ca -- default:	1 [✓ nodes] cpus	03:00:00

Figure 4.3: Placeholder Group Creation (step two)

```
#!/bin/sh

## PBS placeholder template for aurora.nic.ualberta.ca.

## PBS-specific options.
#PBS -S /bin/sh
<--name-->
<--queue-->
<--processors-->
<--maxtime-->
<--merge-->
<--emailaddr-->
<--mailopts-->

## Environment variables:
##   CLS-MACHINE - points to the command-line server’s host.
##   CLS-DIR - remote directory in which the command-line server
##             is located.
## Note the backquote, which executes the quoted command.
JOB=‘/usr/local/bin/ssh $CLS-MACHINE "$CLS-DIR/getcmdline action=nextcmd <--ident--> \
    host=aurora.nic.ualberta.ca jobid=$PBS_JOBID"‘
if [ $? -ne 0 ]; then
    /usr/local/bin/ssh $CLS-MACHINE "$CLS-DIR/getcmdline action=dequeue <--ident--> \
        host=aurora.nic.ualberta.ca jobid=$PBS_JOBID"
    exit 111
fi

## Execute the command from the command-line server.
eval $JOB

## Resubmit the placeholder to "play nice" with the scheduler.
NEWID=‘qsub <--scriptname-->‘
/usr/local/bin/ssh $CLS-MACHINE "$CLS-DIR/getcmdline action=donejob <--ident--> \
    host=aurora.nic.ualberta.ca jobid=$PBS_JOBID newid=$NEWID"
```

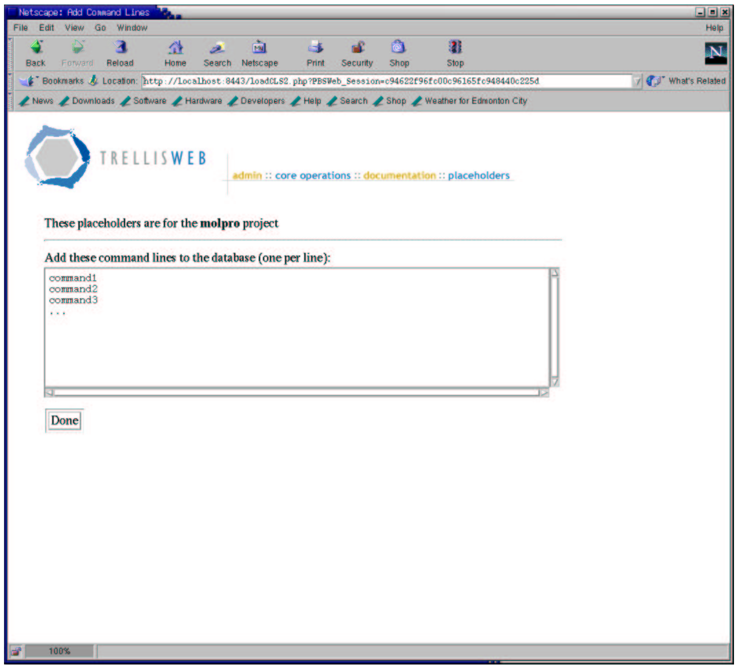
Figure 4.4: Placeholder Template for the University of Alberta MACI SGI Complex

to each of the sites. Placeholder scripts are created from generic templates, one of which is shown in Figure 4.4. A template exists for each site and contains most of the placeholder logic, with the exception of a few stubs (denoted by `<--` ... `-->`). This allows administrators to specialize the templates by providing host-specific file locations (e.g., `/usr/local/bin/ssh`) and other such information. The stubs are replaced at placeholder customization time by parameters provided by the user. In this way, placeholder templates are as reusable as possible.

Finally, placeholder scripts (customized templates) are securely copied to the destination sites and are placed under the user’s remote account in a known location. Once the scripts have been uploaded to their destinations, they are ready to be activated.

4.3.3 Loading the Command-line Server

Before placeholder scheduling is of any use, jobs must be entered into the command-line server database. Although this can be performed using SQL queries, TrellisWeb provides a much easier interface (Figure 4.5). Jobs are entered into the database individually, with the underlying database system providing an ordering to them. Recall that the command-line server may remove jobs in any order, so the underlying order imposed by the database system may or may not be important.


Netscape: Add Command Lines
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8443/loadCLS2.php?PBSWeb_Session=c94622f96fc00c96165fc948440c225d What's Related
News Downloads Software Hardware Developers Help Search Shop Weather for Edmonton City
TRELLISWEB
admin :: core operations :: documentation :: placeholders
Done
100%

These placeholders are for the **molpro** project

Add these command lines to the database (one per line):

```
command1
command2
command3
...
```

Figure 4.5: Loading the Command-line Server

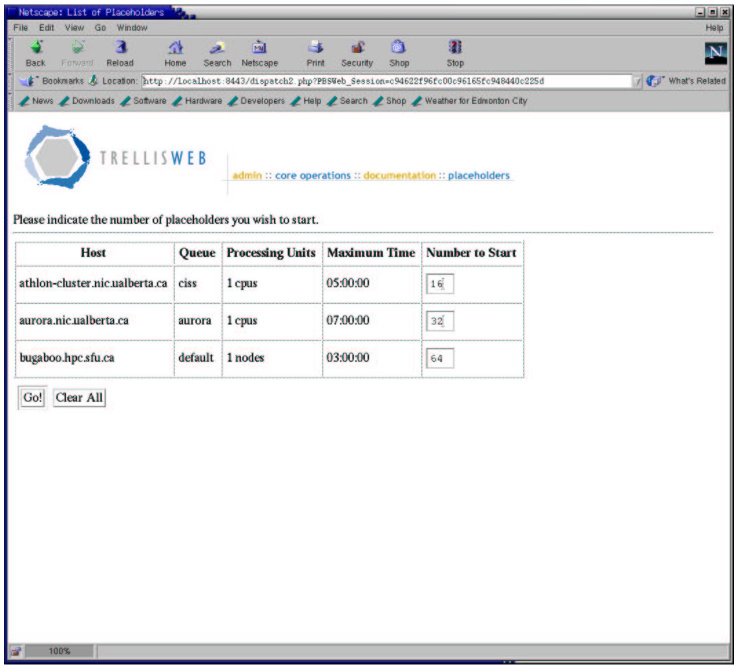
4.3.4 Dispatching Placeholders

Once job information has been entered into the command-line server database, the placeholders are ready to begin execution. Placeholders of a group are typically launched all at once so that the overall task may be started as a whole. This adds to the illusion that the placeholder group represents a single task on a single computer. However, the user must decide how many placeholders to submit at each site, a decision that must be made with some prior knowledge of the site. If the site utilizes a batch scheduling system that is capable of queuing additional jobs, any detrimental effects of dispatching too many placeholders to a site is reduced.

Figure 4.6 shows the main step in dispatching placeholders. Previous information, provided when the placeholder scripts were created, is again presented to the user to aid in recalling the specifics of the site. It is then up to the user to specify the number of placeholders to dispatch at each location. All placeholders dispatched to a given location are identical (provided they are part of the same group). If this is not the desired effect, additional placeholders with different parameters may be added to the same location via another TrellisWeb page.

4.3.5 Monitoring and Managing Placeholders

After the placeholders have been dispatched, all that remains is to monitor their progress. TrellisWeb provides an integrated view of placeholder progress on a per-group basis. Each placeholder in a group may be in one of two states at any given point in time. A placeholder displayed as being in the


Netscape: List of Placeholders
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8443/dispatch2.php?PBSWeb_Session=c94622f96fc00c96165fc948440c225d
What's Related
News Downloads Software Hardware Developers Help Search Shop Weather for Edmonton City
TRELLISWEB
admin :: core operations :: documentation :: placeholders
100%

Please indicate the number of placeholders you wish to start.

Host	Queue	Processing Units	Maximum Time	Number to Start
athlon-cluster.nic.ualberta.ca	ciss	1 cpus	05:00:00	16
aurora.nic.ualberta.ca	aurora	1 cpus	07:00:00	32
bugaboo.hpc.sfu.ca	default	1 nodes	03:00:00	64

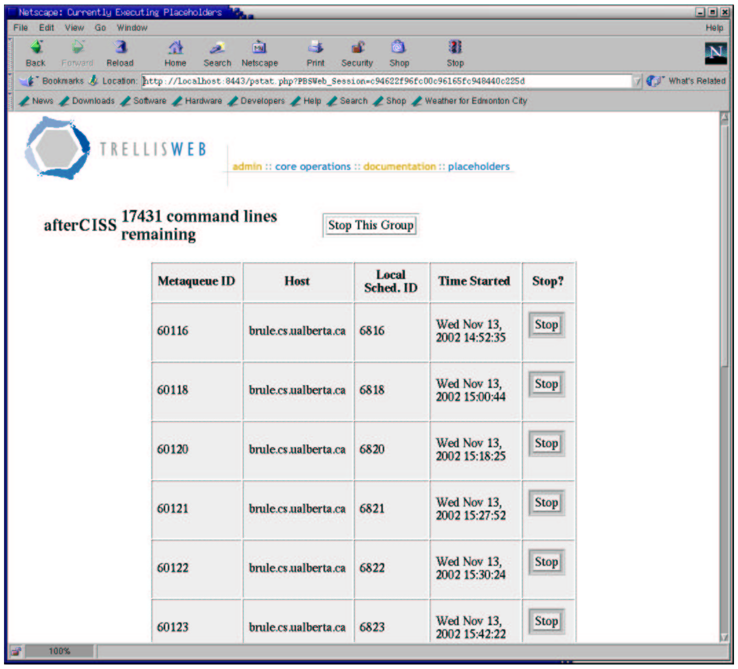
Figure 4.6: Placeholder Dispatch

---

queued state is one that has been dispatched but has not yet contacted the command-line server for work. Most likely, the placeholder has been queued by an underlying batch scheduler. Alternatively, the placeholder may be in the *executing* state. A placeholder is considered to be in this state if it has received work and has not yet reported back that the work is done.

The monitoring page of TrellisWeb is shown in Figure 4.7. The user is presented with group information such as the name of the group and the number of jobs remaining in the command-line server database. This information can be interpreted by the user to indicate the progress of the overlay metacomputer. Also provided is per-placeholder information such as the state of the placeholder (described previously), the local scheduler identifier (assigned by the underlying scheduling system), and the time when the placeholder began executing. The user may elect to stop the entire group of placeholders, or one placeholder at a time. Stopping a placeholder effectively kills it by interacting with the underlying batch scheduler (in the case where one exists) or issuing a `kill(1)` command (on zero-infrastructure systems).

Additional placeholders may be dispatched to augment currently running ones. These placeholders may be pre-existing members of the group, or new ones added on-the-fly. Jobs may also be added to the command-line server database at any point prior to, during, or after placeholder dispatch to provide maximum flexibility for the user. The user may terminate some or all of the placeholders should he/she decide to stop execution. In effect, the user may interact with the overlay metacomputer as a whole or in pieces.


Netscape: Currently Executing Placeholders
File Edit View Go Window Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://localhost:8443/pstat.php?PBSWeb_Session=c94622f96fc00c96165fc948440c225d What's Related
News Downloads Software Hardware Developers Help Search Shop Weather for Edmonton City
TRELLISWEB
admin :: core operations :: documentation :: placeholders
Stop This Group
100%

afterCISS 17431 command lines remaining

Metaqueue ID	Host	Local Sched. ID	Time Started	Stop?
60116	brule.cs.ualberta.ca	6816	Wed Nov 13, 2002 14:52:35	Stop
60118	brule.cs.ualberta.ca	6818	Wed Nov 13, 2002 15:00:44	Stop
60120	brule.cs.ualberta.ca	6820	Wed Nov 13, 2002 15:18:25	Stop
60121	brule.cs.ualberta.ca	6821	Wed Nov 13, 2002 15:27:52	Stop
60122	brule.cs.ualberta.ca	6822	Wed Nov 13, 2002 15:30:24	Stop
60123	brule.cs.ualberta.ca	6823	Wed Nov 13, 2002 15:42:22	Stop

Figure 4.7: Placeholder Monitoring and Management

4.4 Additional Command-line Server Functionality

The simple functionality of the command-line server presented in Section 3.4 is adequate for manual placeholder scheduling, but is insufficient to allow for all features of TrellisWeb. As a result, the command-line server used with TrellisWeb is slightly more complex than the bare-bones version previously described. This increased complexity, however, does not detract from the core behaviour of the command-line server, which still adheres to the required design.

The major change to the command-line server is the ability to record when placeholders start and finish executing jobs. The current state of placeholder execution is recorded in the database, and requires that the placeholder invoke the command-line server upon job completion. This additional invocation requires approximately double the overhead of a simple command-line server implementation, but opens the door to a number of additional benefits such as fault tolerance and better placeholder management by the user. Without the state information, a user (and indeed the fault tolerance system) would be unable to determine what progress has been made. Although this is sufficient for a number of applications, the additional information provided by the TrellisWeb command-line server is invaluable because of the potential benefits it provides.

The TrellisWeb command-line server can also log information about completed jobs. Users may wish to review the completed jobs and the times when they were run, as well as how much time was consumed. Administrators may wish to review usage trends for accounting or other administrative purposes. The increased complexity for logging is transparent to the placeholders, and only requires additional database accesses by the command-line server.

4.5 Simple Fault Tolerance

Due to the potential size of an overlay metacomputer, faults can become a problem. As the overlay metacomputer becomes larger and larger, so too does the potential for placeholder failures. Although some applications may be inherently fault tolerant, most are not. The level of fault tolerance provided by TrellisWeb is minimal, but is enough to prevent the need for manual identification and resubmission of failed jobs.

Placeholder scheduling has many advantages with respect to fault tolerance. Because placeholders work as a team to complete a set of jobs, the failure of one or a few placeholders is not a catastrophic event. The performance of placeholder scheduling in such a situation gracefully declines as placeholders are lost to failure. This ability comes purely from the fact that placeholders work on a pull model and are often used simultaneously in groups to load balance work across multiple sites.

The basic idea behind fault tolerance in TrellisWeb, beyond that inherent in placeholder scheduling, is three-fold. The first opportunity for failure recovery comes when placeholders attempt to use certain utilities (such as SSH) that may fail due to transient conditions. To compensate for such failures, placeholders in TrellisWeb retry such utilities up to five times with a waiting period in between. The second opportunity comes when connections to the command-line server time out due to an error at the command-line server machine. In such an event, the command-line server script used with TrellisWeb fails after a timeout. Furthermore, it fails in such a way that the placeholder behaves as if the connection failed entirely. Finally, the third opportunity comes when a placeholder is identified as failed, and is rescheduled by the TrellisWeb fault tolerance script. Each of these three levels, in addition to the inherent fault tolerance of placeholder scheduling, make placeholder scheduling tolerant to many different types of common failures.

The logging features of TrellisWeb record information such as when a placeholder was submitted and when a placeholder begins executing a job. With this information, it can be determined when a job is suspect of failure. The simple fault tolerance provided by TrellisWeb imposes an upper limit on time spent by a placeholder in the queued and executing states. Should a placeholder exceed the limit, the fault tolerance script will identify the placeholder as failed and restart it. Pseudocode of the fault tolerance script is shown in Figure 4.8.

Two possible types of failures are accounted for by the fault tolerance script. First, the set of currently executing jobs is examined. Should any of the currently executing jobs be executing for too long (defined by `EXEC-FAILURE`), the job is considered to have failed (line 5). Because the potential exists for certain jobs to contain faulty information that causes failures, a threshold of retries is maintained (`MAX-RETRIES`). Jobs are retried up to `MAX-RETRIES` and then abandoned as failed (lines 7-11). Finally, the corresponding placeholder must be restarted (lines 12-15), as this

```
1  EXEC-FAILURE := 12 hours
2  QUEUE-FAILURE := 24 hours
3  MAX-RETRIES := 3
4
5  failed := set of executing placeholders with start time > EXEC_FAILURE ago
6  for each placeholder in failed do
7      if placeholder.job.retries <= MAX_RETRIES then
8          placeholder.job.retries := placeholder.job.retries + 1
9          reinsert the job into the list of waiting jobs
10     else
11         insert the job into the failed list
12     delete the placeholder via the underlying scheduling system (or kill it if ZINF)
13     remove the old placeholder from the database
14     resubmit the placeholder to the underlying scheduling system
15     enter the placeholder into the database as queued
16 done
17
18 failed := set of queued placeholders with queue time > QUEUE_FAILURE ago
19 for each placeholder in failed do
20     delete the placeholder via the underlying scheduling system (or kill it if ZINF)
21     remove the old placeholder from the database
22     resubmit the placeholder to the underlying scheduling system
23     enter the information into the database as queued
24 done
```

Figure 4.8: Pseudocode TrellisWeb Fault Tolerance Script

is potentially the only placeholder in the group that is executing. A similar process is carried out for queued placeholders (lines 18-24).

In addition to the central fault tolerance implemented on the database side, placeholders themselves provide a small amount of fault tolerance. Because of the possibility of transient network faults, SSH connections may fail on occasion. If such a failure were to happen whilst a placeholder was attempting to contact the command-line server, it may result in the loss or corruption of job information, or possibly cause the placeholder to believe that no more work is available. To prevent this, all SSH connections made by placeholders are retried five times upon failure. A configurable period of ten minutes between tries is imposed to allow connections to be re-established and correct operations to resume. This gives the placeholder a 50 minute buffer for any failures that would cause SSH to report an error (both network and non-network related).

Of course, TrellisWeb fault tolerance does not preclude the use of other fault tolerant systems or practices. In fact, fault tolerance at the application level is encouraged as the simple fault tolerance introduced here is incapable of detecting all possible errors. TrellisWeb fault tolerance is intended to provide minimal functionality to those applications that have little or none.

4.6 Concluding Remarks

The primary goal of TrellisWeb is to make placeholder scheduling a more useful and practical tool for metacomputing. Placeholder scheduling is flexible with respect to implementation. Some users may take advantage of this flexibility by implementing placeholder scheduling themselves. On the other hand, most users want something that is easy to use while still providing the benefits of metacomputing.

TrellisWeb trades some of the complexity and flexibility of placeholder scheduling for ease-of-use. The web-based interface restricts users to the most common functions of placeholder scheduling, both to reduce the difficulty of using placeholders and to prevent users from misusing placeholder scheduling. However, TrellisWeb is still flexible enough to perform placeholder scheduling for most intended applications. In the future, the command-line server may be selectable so that the scheduling policy may be modified (e.g., dependency-based scheduling).

The inclusion of fault tolerance in TrellisWeb augments existing application-level fault tolerance, if such fault tolerance exists. At the same time, TrellisWeb provides a minimal level of fault tolerance to users who are unwilling or unable to add fault tolerance to their applications. Because of the larger scope of metacomputers beyond that of single-site computers, faults must be expected and hence must be addressed at least in some basic way.

Chapter 5

Empirical Evaluation

Placeholder scheduling was designed as a mechanism for scheduling jobs across multiple computational resources that may or may not be shared with other users. Placeholder scheduling can be verified by the use of live execution runs, simulation-based analysis, or trace data. Live execution runs were chosen because they deal with a realistic instead of artificial load (as in simulation data), and no collection mechanism is required to collect trace data *a priori*. Recall that placeholder scheduling is largely a mechanistic approach to metacomputing and that the current policy used by placeholder scheduling (FCFS) is relatively simple, so the results presented in this chapter are not particularly interesting from a policy standpoint.

Placeholder scheduling is primarily interested in providing low makespans for a set of jobs based on good load balancing. Along with low makespans, utilization of the participating sites is increased with respect to the number of placeholders present because no placeholder idles while capable of performing work. Any metacomputing system that allows a user to perform computations at more than one site is beneficial, but performance should increase in relation to the size of the metacomputer. Improved performance is perhaps the single greatest reason for using a metacomputer in the first place. As a result, the performance of placeholder scheduling will be the metric of its usefulness.

5.1 Experiment #1: Proof-of-Concept

The basic ideas behind placeholders and placeholder scheduling are fairly straightforward: centralize the jobs of the workload into a metaqueue (i.e., the command-line server), use placeholders to pull the job to the next available local queue, and use late binding to give the system maximum flexibility in job placement and load balancing. Such a system is built using only widely-available and deployed infrastructure, such as SSH, and existing batch scheduling systems, such as PBS. The goal of the proof-of-concept experiment is to validate that such a system can be built.

Empirical evidence derived from this experiment shows that placeholder scheduling can load balance across multiple multiprogrammed systems with independent queues. The experimental platform consists of machines within the same administrative domain; a more controlled and uniform

Total Number of Jobs	Scheduler	See Fig.	Makespan (Real-Time Load Imbalance)	Time When System Becomes Idle (number of jobs completed) [units of work completed] A	B	C
(a) 50	User script, Multiprogrammed	5.2, 5.3	13,981 (72%)	3,928 (17 jobs) [1,000 units]	13,981 (16 jobs) [1,025 units]	3,939 (17 jobs) [1,075 units]
(b) 50	Placeholder, Multiprogrammed	5.4, 5.5	5,793 (14%)	5,204 (24 jobs) [1,325 units]	5,793 (6 jobs) [425 units]	4,984 (20 jobs) [1,350 units]
(c) 200	Placeholder, Multiprogrammed	5.6, 5.7	17,801 (1.9%)	17,460 (81 jobs) [4,450 units]	17,801 (24 jobs) [1,300 units]	17,560 (95 jobs) [5,250 units]
(d) 200	Placeholder, Uniprogrammed	5.8, 5.9	17,053 (2.1%)	16,858 (78 jobs) [4,300 units]	17,053 (26 jobs) [1,250 units]	16,692 (96 jobs) [5,450 units]

Table 5.1: Summary of Proof-of-Concept Results

Makespan is generally defined as the total real time required to complete the workload, from the arrival of all the jobs at time zero until completion of the last job. *Real-Time Load Imbalance* is defined here as the time difference between when the last system becomes idle and when the first system becomes idle, then normalized by the makespan. (i.e., Real-Time Load Imbalance $= \frac{last\ system\ idle\ time - first\ system\ idle\ time}{makespan}$, in which $makespan = time\ workload\ finished - time\ workload\ started$.) The goal is to minimize makespan and real-time load imbalance for a given workload.

Recall that individual jobs vary in the required units of work. In (a), System B completes fewer jobs than System A, but those jobs require more units of work. In (b), System A completes more jobs than System C, but both systems complete nearly the same units of work. All systems finish within 809 seconds (i.e., 14% of the makespan) of each other when using placeholder scheduling. The more jobs in the workload, the greater the opportunity to load balance so as to minimize the makespan and real-time load imbalance. In (c), all systems finish within 341 seconds (i.e., 1.9% of the makespan) of each other. In (d), all systems finish within 361 seconds (i.e., 2.1% of the makespan) of each other.

environment is required to show that placeholder scheduling works. The subsequent experiments examine placeholder scheduling in the context of multiple administrative domains.

5.1.1 User Scripts

To illustrate the pitfalls of the user script approach, in which a user statically assigns jobs to computers, a synthetic workload of 50 jobs was created(Table 5.1(a)) executing on three different workstations: one four-way multiprocessor and two uniprocessors with differing computational power (Table 5.2). Each job is a simple sort program that executes in parallel on the multiprocessor and sequentially on the uniprocessors. The binary executables are built specifically for the machine architecture, the operating system, and the algorithm used (i.e., parallel or sequential), but the over-

System	Description	Sorting Algorithm
A (`caslan`)	SGI Origin 2100, 4 × 350 MHz R12000, 1 GB RAM, Irix 6.5	Parallel
B (`lacrete`)	Single Pentium II, 400 MHz, 128 MB RAM, Linux 2.2.16	Sequential
C (`st-brides`)	Single AMD Athlon XP 1900+, 1.6 GHz, 256 MB RAM, Linux 2.4.9	Sequential

Table 5.2: Experimental Platform: Three Independent Systems and PBS Execution Queues

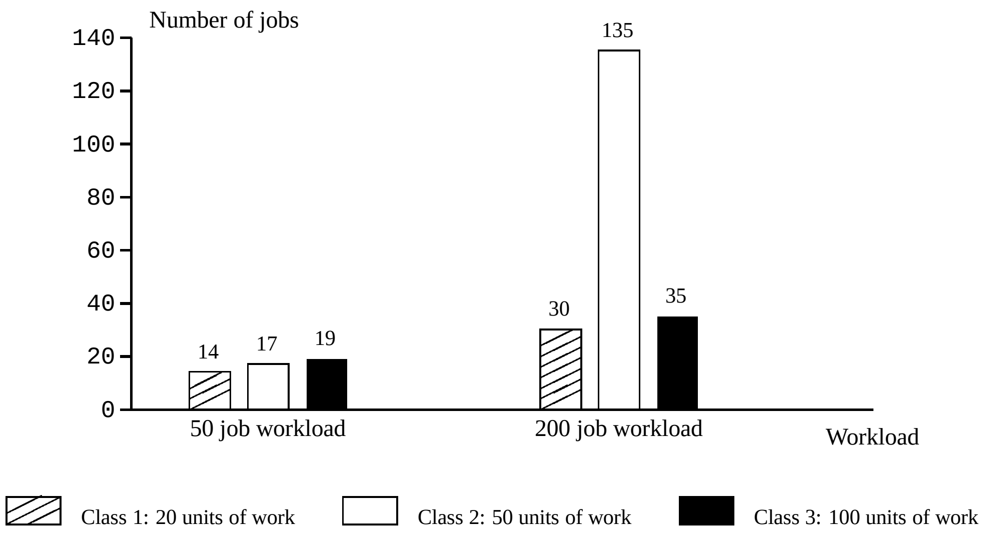


Figure 5.1: Synthetic Workloads: Service-Time Distribution of Jobs

all application is the same over the entire overlay metacomputer (and has the same name on all machines).

Jobs vary in the amount of computation required for the sort. All jobs sort approximately four million keys, but a variable command-line argument controls how many times the sort is repeated for each job (i.e., number of iterations). One iteration is defined as one unit of work for this experiment. Overall, there are 14 jobs with 25 units of work, 17 jobs with 50 units of work, and 19 jobs with 100 units of work (Figure 5.1). Jobs from each class are randomly interleaved in the workload.

Makespan is generally defined as the total real time required to complete the workload, from the arrival of all the jobs at time zero until completion of the last job. *Real-time load imbalance* is defined here as the time difference between when the last system becomes idle and when the first system becomes idle, then normalized by the makespan (i.e., real-time load imbalance $= \frac{last\ system\ idle\ time - first\ system\ idle\ time}{makespan}$, in which $makespan = time\ workload\ finished - time\ workload\ started$). The goal is to minimize makespan and real-time load imbalance for a given workload.

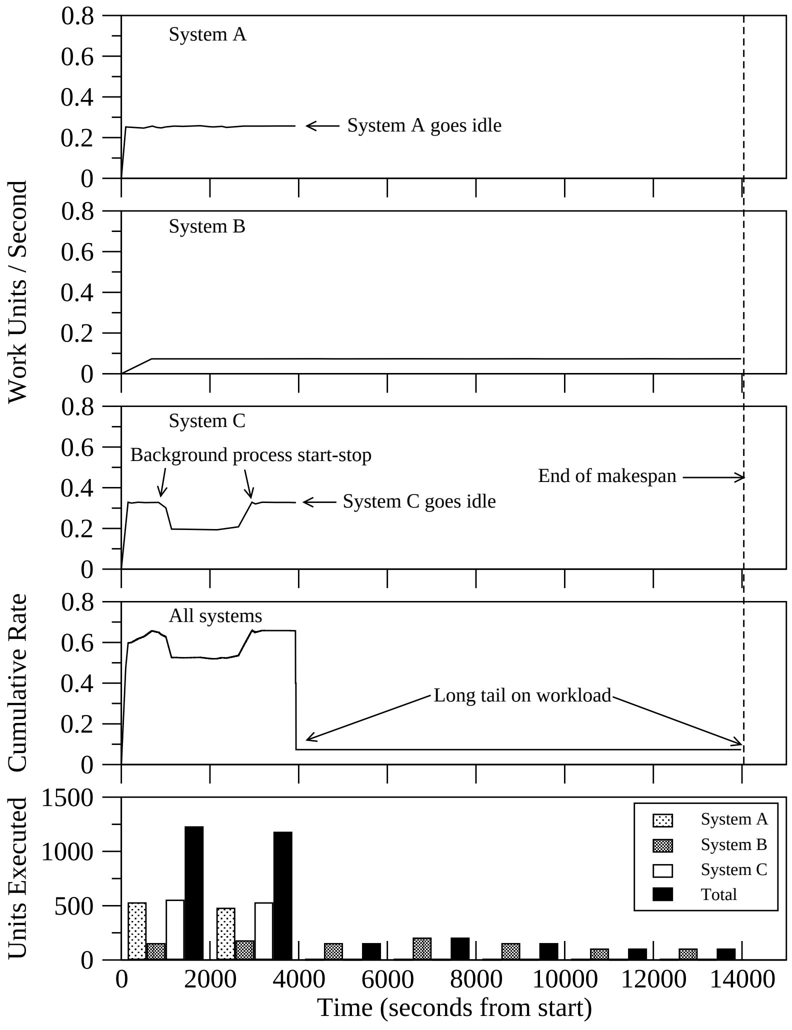


Figure 5.2: Throughput with User Scripts, Multiprogrammed, 50 Jobs
(Also see Table 5.1(a). Poor load balancing due to slowness of System B results in a long tail on the makespan of the workload.)

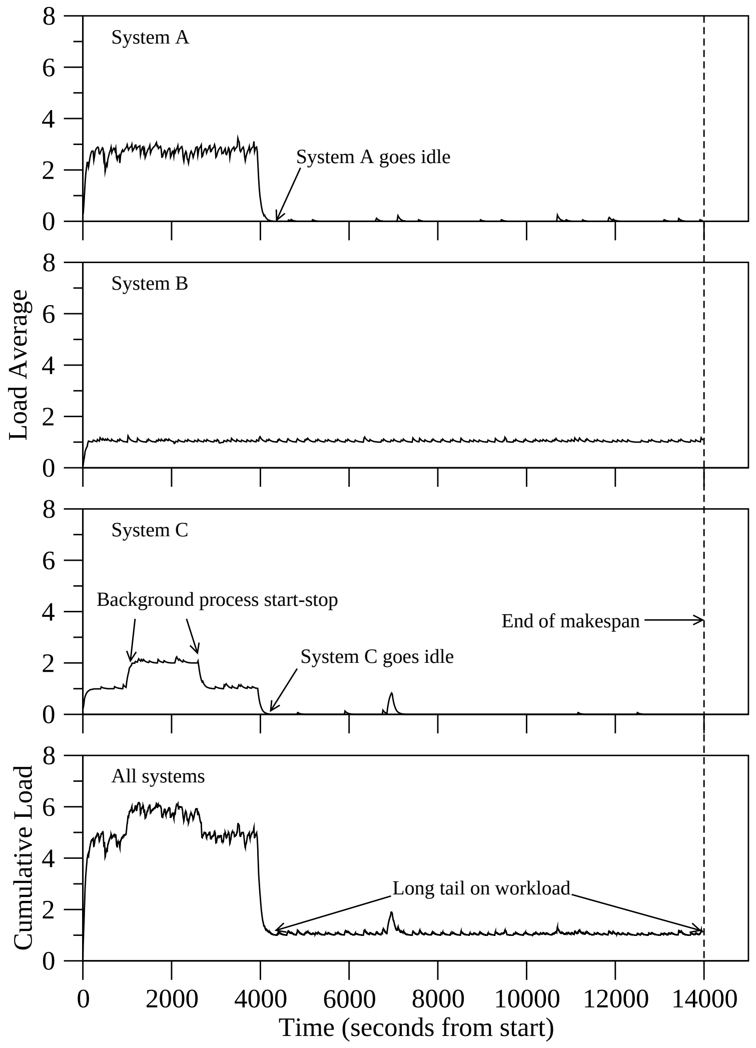


Figure 5.3: System Load with User Scripts, Multiprogrammed, 50 Jobs

A trace of the start and finish times for each job is kept during the makespan of the workload. The load on each of the three systems is also monitored using the Unix `uptime` command. After post-processing the traces, the throughput of the three systems (Figure 5.2) and how throughput varies with the observed load (Figure 5.3) was examined. The figures represent a single execution of the workload. Of course, different executions of the workload will have small variations, but all of the major factors (e.g., other users, other compute-intensive processes) are controlled and the traces presented are representative of the workload.

For static load balancing using the user scripts, Systems A and C are given 17 jobs each, and System B is given 16 jobs. Both Systems A and C finish their jobs within 4,000 seconds of starting (Table 5.1(a)). Figure 5.2 shows the measured throughput for one particular execution of the workload. The upper three graphs show the normalized job throughput on Systems A, B, and C, respectively. The fourth graph is the cumulative throughput of Systems A, B, and C. The bottom-most graph shows the total units of work completed by each system for each 2,000 second interval in the makespan.

Figure 5.3 shows the observed load on each of the systems (top three graphs) and the cumulative load on all the systems (bottom-most graph) for the makespan. There are natural fluctuations present in the system load due to the nature of the `uptime` command and the presence of operating system daemons on the systems. Each of the systems execute one job at a time, with System A executing a parallel version of the sorting program that uses all four processors.

For this experiment, Systems A and B are not shared with other jobs or processes. The entire load on Systems A and B is due to our workload (Figure 5.3). However, a background process is artificially introduced on System C from time 830 seconds (approximately) to 2600 seconds. The background process competes for the CPU and approximates the impact of other users in a multiprogrammed environment.

The four processors of System A provide throughput slightly below that of the much faster single processor of System C (i.e., 0.25 units of work/second versus 0.325 units of work/second) (Figure 5.2). System B has the slowest processor and a throughput of approximately 0.07 units of work/second. When the background process on System C competes for the CPU, System C's throughput drops to 0.2 units of work/second. The impact of the background process on System C can also be seen in Figure 5.3: the load increases from 1.0 to 2.0.

Although System C has to compete with another process for part of the workload, its greater computational power allows it to complete its jobs in 3,939 seconds versus 3,928 seconds for System A. However, System B is substantially slower than the other systems and it requires 13,981 seconds to finish all of its assigned jobs. Therefore, System B represents a slow system in this experiment and the makespan of the workload is 13,981 seconds with a real-time load imbalance is a 72%. Of course, these empirical results will change as the specific workload and the experimental platform are varied.

To solve the problem of poor load balancing and long makespans with user scripts, the user could make some changes. For example, if the user knows in advance that System B is the slowest, the user scripts can be modified so that Systems A and C are initially given more jobs. Also, the user could monitor the three systems so that when Systems A and C go idle, they may take on some of the remaining jobs for System B.

Ideally, a scheduler should automatically adapt to slow systems and queues by load balancing. After all, if an overlay metacomputer has dozens of sites, it is not realistic to expect the user to deal with slow systems. Furthermore, other jobs and users may dynamically cause one system or queue to be slower, which further complicates load balancing given a user script strategy.

Unfortunately, if the system administrators do not or cannot provide another form of metaqueue or computational grid, the user (currently) has no choice other than to manually place and load balance their jobs across sites. Fortunately, placeholder scheduling can be implemented directly by the user, without support from system administrators. It is secure, and achieves the same benefits as other forms of metaqueues.

5.1.2 Placeholders: Example with 50 Jobs

As seen in the previous example with user scripts, load balancing is a serious problem when dealing with heterogeneous hardware platforms and in the presence of other users and jobs. When placeholder scheduling is used for the same 50 job workload as was used in Section 5.1.1, a makespan of 5,793 seconds and a real-time load imbalance of 14% are obtained. (Table 5.1(b), Figure 5.4, and Figure 5.5). The bottom bar graph of Figure 5.4 shows the per-system and total amount of work performed for each 2,000 second interval.

Once again, each system contributes to the throughput according to its individual computational power, and a competing background process is present on System C for part of the makespan. Again, System C finishes first (at 4,984 seconds) when there are no more jobs. However, no system goes idle if there is still a piece of work that has not already been allocated to a system. Of course, it is difficult for all three systems to finish at exactly the same time, because the amount of computation required for the job on each system can vary. Nonetheless, the three systems finish within 809 seconds (i.e., 14% of the makespan) of each other.

It is *not* a new result to be able to dynamically load balance a workload across multiple systems. Load balancing and the optimization of throughput shown for placeholders and overlay metacomputers can also be achieved using other forms of metaqueues and computational grids. What is new is how placeholders can also provide this important functionality and how they can be implemented in situations in which other metaqueues and computational grids do not yet exist.

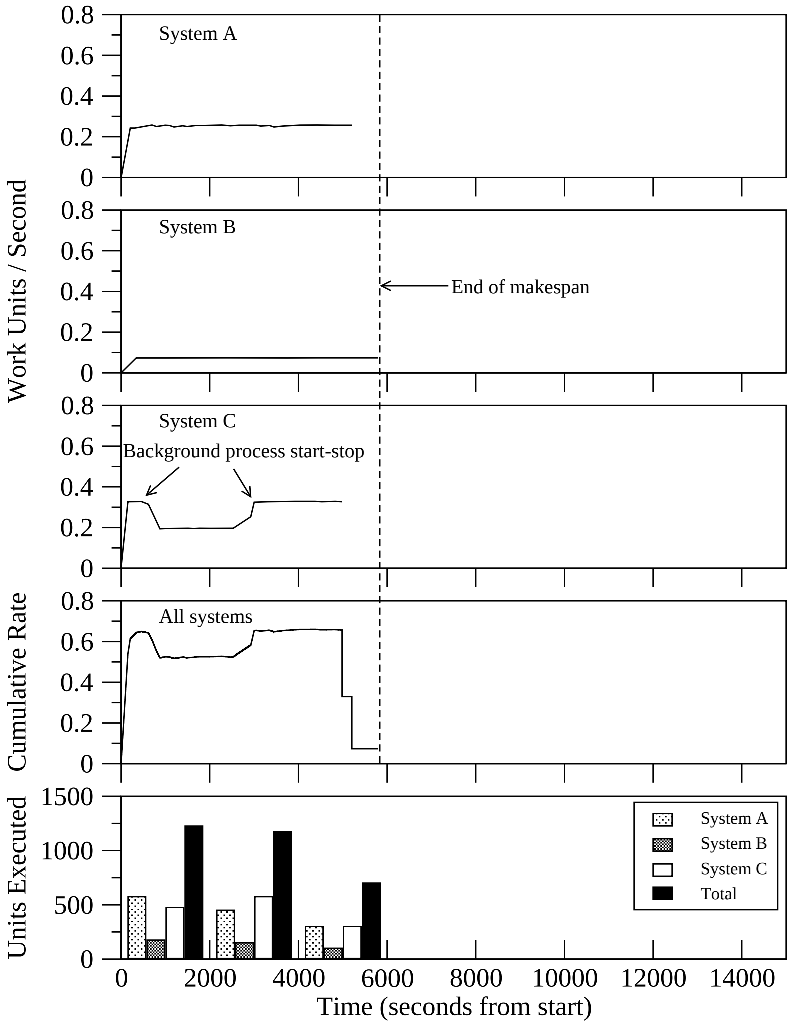


Figure 5.4: Throughput with Placeholder Scheduling, Multiprogrammed, 50 Jobs
(Also see Table 5.1(b). Better load balancing with placeholders results in a smaller makespan, despite the competition from a background process on System C.)

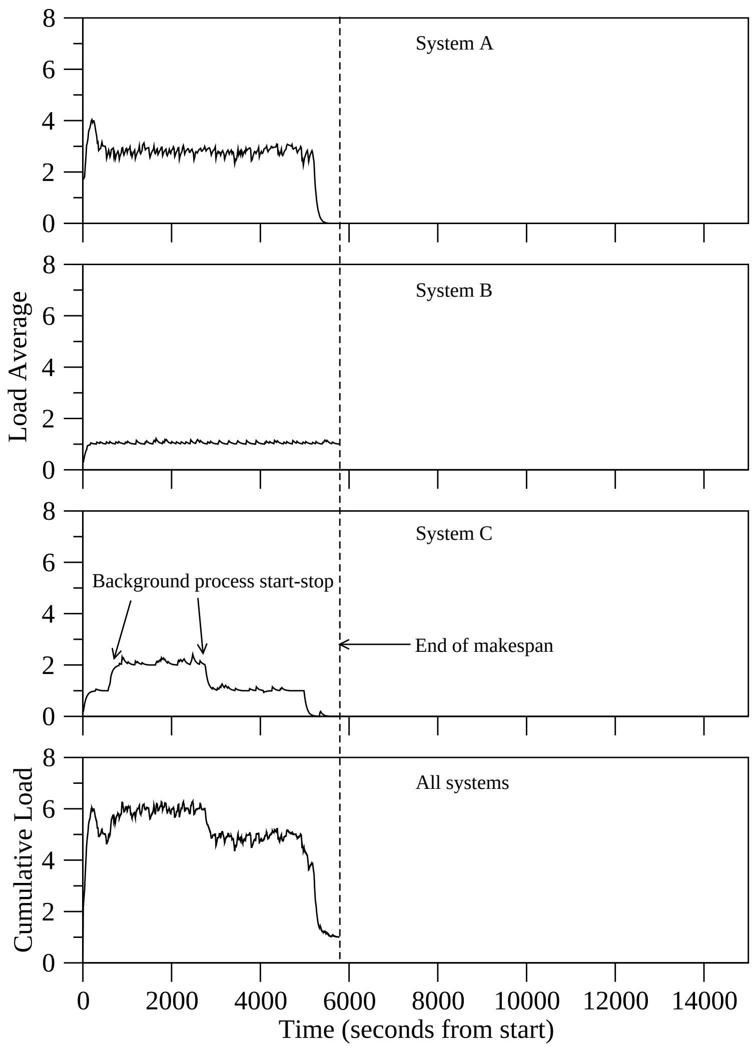


Figure 5.5: System Load with Placeholder Scheduling, Multiprogrammed, 50 Jobs

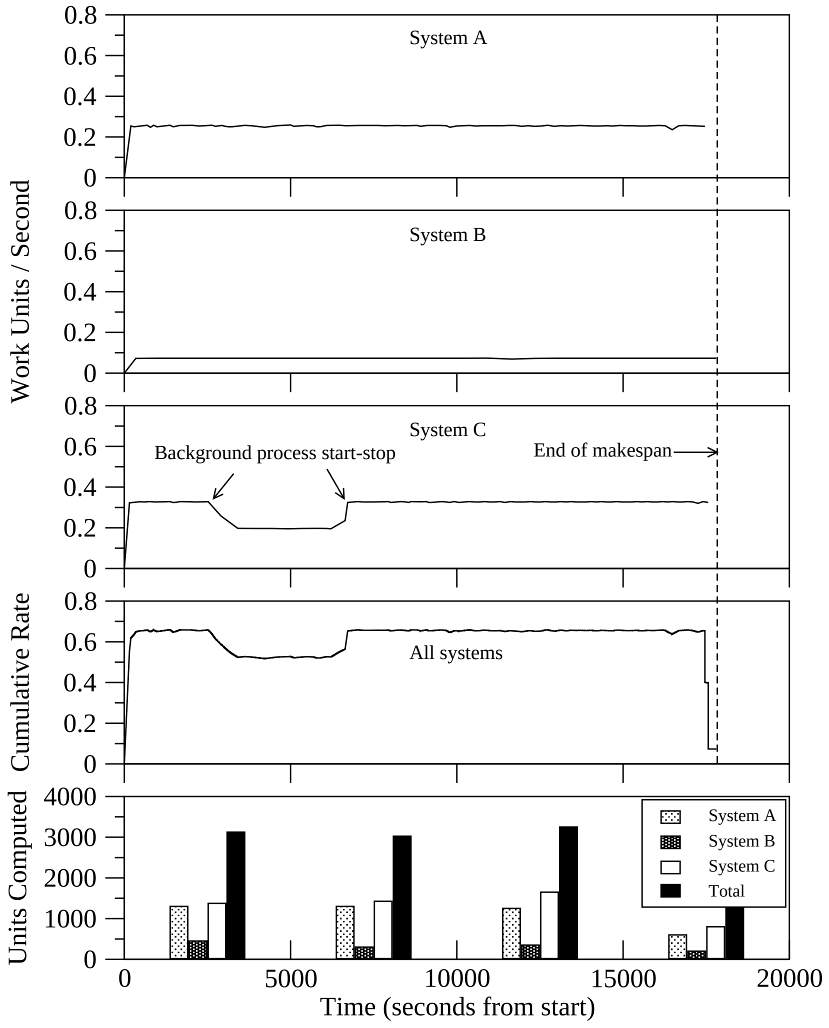


Figure 5.6: Throughput with Placeholder Scheduling, Multiprogrammed, 200 Jobs
(Also see Table 5.1(c). Load balancing with placeholders results all systems going idle at nearly the same time. Real-time load imbalance is 1.9% of the makespan.)

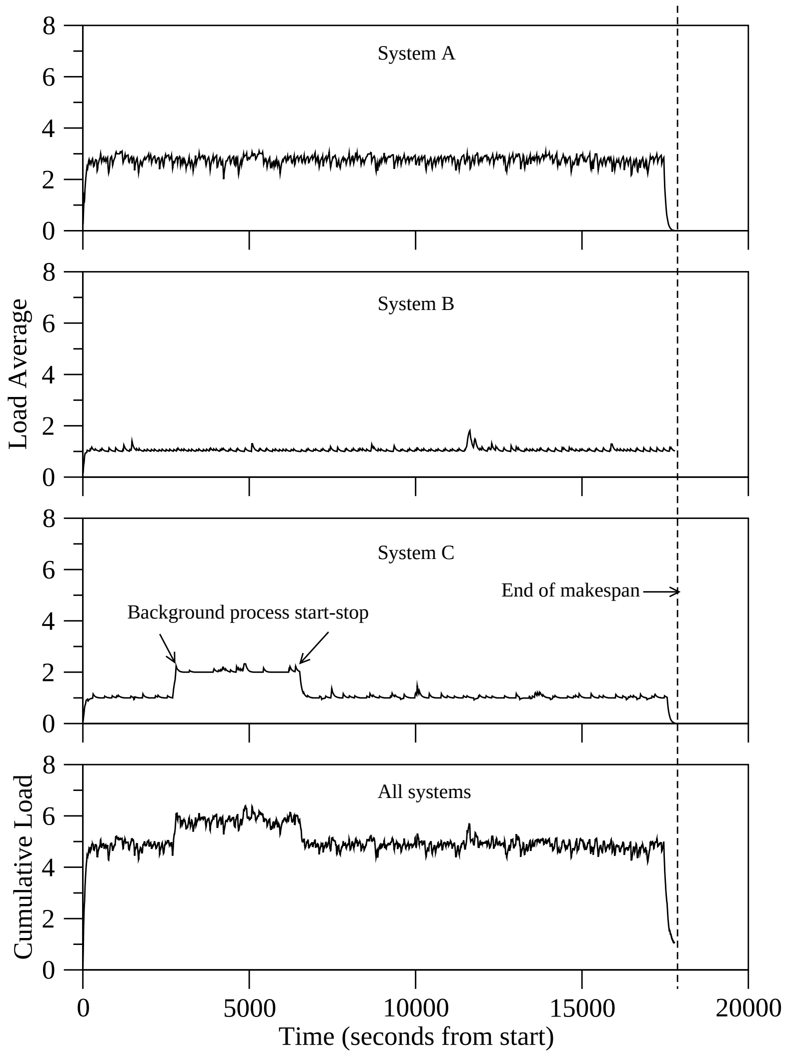


Figure 5.7: System Load with Placeholder Scheduling, Multiprogrammed, 200 Jobs

5.1.3 Placeholders: Example with 200 Jobs

An experiment similar to that of Section 5.1.2 was performed with placeholder scheduling using a total workload of 200 jobs, instead of only 50. Within the workload there were 30 jobs with 25 units of work, 135 jobs with 50 units of work, and 35 jobs with 100 units of work (Figure 5.1). The initial 100 jobs of the workload contained a random mix of 25, 50, and 100 units of work, and the final 100 jobs contained 50 units of work each. For this workload, a makespan of 17,801 seconds was obtained. (Table 5.1(c), Figure 5.6, and Figure 5.7). The bottom bar graph of Figure 5.6 shows the per-system and total amount of work performed for each 5,000 second interval. Once again, a competing background process is artificially introduced on System C during the makespan.

Given the larger workload, placeholder scheduling is able to better load balance across the three systems such that they go idle at nearly the same time (i.e., 1.9% of the makespan). The fact that the last 100 jobs in the metaqueue have a uniform 50 units of work helps in the load balancing, but there is a mix of jobs, with different service times, for the first 100 jobs.

5.1.4 Placeholders: Example with 200 Jobs, Uniprogrammed

For a final experiment, the same 200 jobs as before (Section 5.1.3) are used. This time, Systems A, B, and C are dedicated to the workload; there are no competing users or processes. Consequently, a lower makespan of 17,053 seconds was obtained (Table 5.1(d), Figure 5.8, and Figure 5.9). The bottom bar graph of Figure 5.8 shows the per-system and total amount of work performed for each 5,000 second interval. The only difference between this experiment and the one in Section 5.1.3 is the absence of a competing process on System C during the workload.

Not surprisingly, the makespan in the uniprogrammed case is lower than in the multiprogrammed case, given that the entire computational power of all three systems is devoted to the workload. However, the relative difference in makespans is small (i.e., 4.3%, 17,053 seconds versus 17,801 seconds) because placeholder scheduling is effective in balancing the load in the multiprogrammed case that CPU cycles lost to a competing process are compensated for by the remaining two systems.

5.1.5 Summary of Proof-of-Concept Experiments

Based on the experiments and synthetic workloads, we can see:

1. User scripts and static load balancing are problematic on the heterogeneous and shared computer systems likely to be found in a grid or metacomputer environment. (Section 5.1.1 and Table 5.1(a).)

2. Placeholder scheduling effectively implements dynamic scheduling and load balancing in situations in which another form of metaqueue or grid is not, or cannot, be implemented. The improvement in makespan and real-time load imbalance, as compared to user scripts, is substantial. (Sections 5.1.2 and 5.1.3 and Table 5.1(b),(c).)

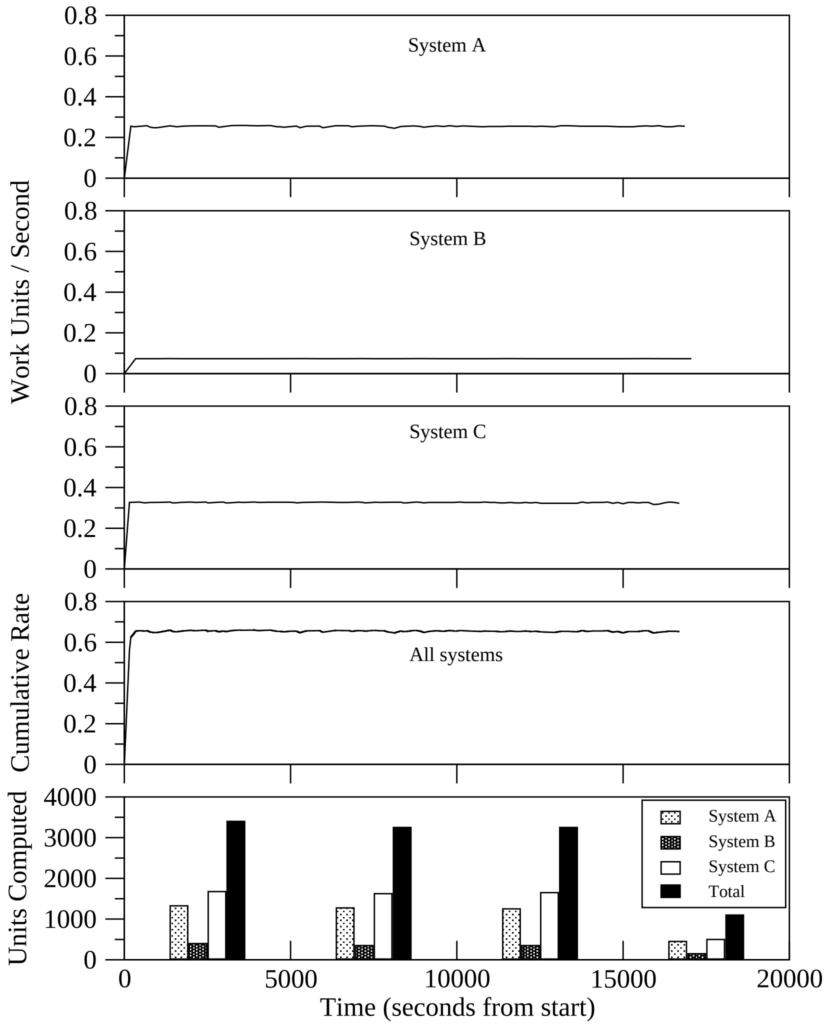


Figure 5.8: Throughput with Placeholder Scheduling, Uniprogrammed, 200 Jobs
(Also see Table 5.1(d). Real-time load imbalance is 2.1% of the makespan.)

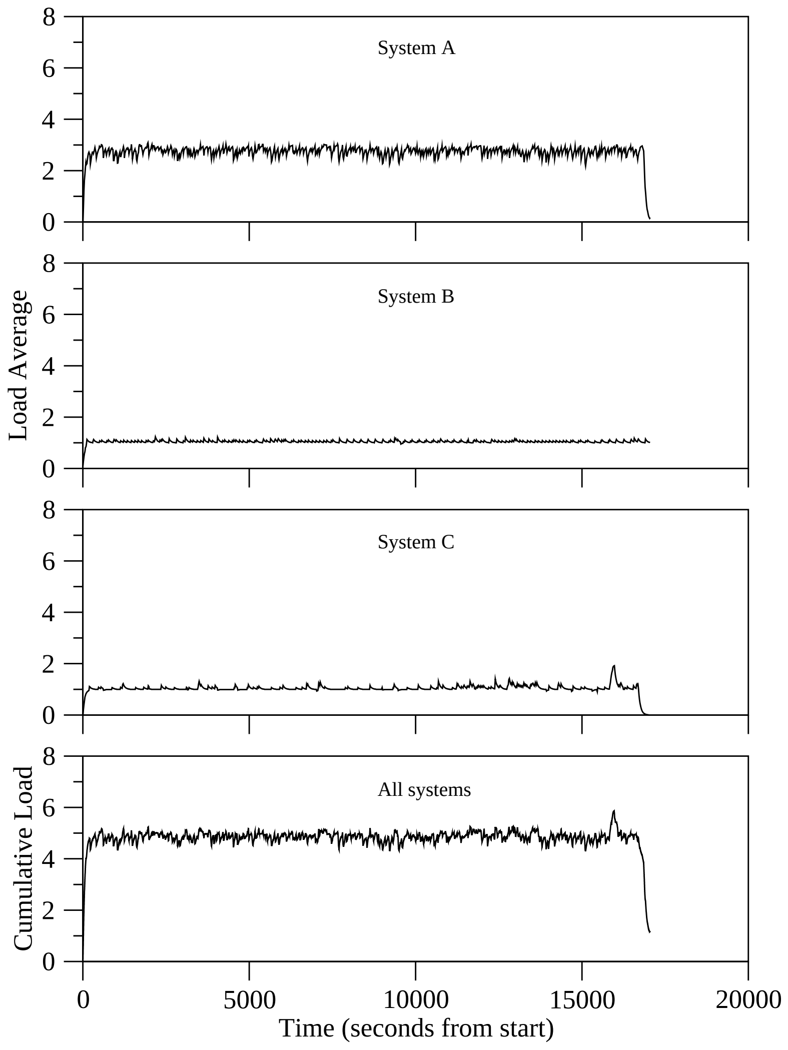


Figure 5.9: System Load with Placeholder Scheduling, Uniprogrammed, 200 Jobs

System	Description	Interconnect	Scheduler	Method
A (`aurora`)	SGI Origin 2000, 46 × 195 MHz R10000, 12 GB RAM, Irix 6.5.14f	Shared Memory NUMA	PBS	Parallel Shared Memory
B (`lacrete`)	Single Pentium II, 400 MHz, 128 MB RAM, Linux 2.2.16	None	Sun Grid Engine	Sequential
C (`maci-cluster`)	Alpha Cluster, mixture of Compaq XP1000, ES40, ES45, and PWS 500au, 206 processors in 122 nodes, each node has from 256 MB to 8 GB RAM, Tru64 UNIX V4.0F	Gigabit Ethernet	PBS	Parallel Distributed Memory (i.e., MPI)

Table 5.3: Experimental Platform for the Parallel Sorting Application

3. Placeholder scheduling is also effective in dealing with multiprogrammed environments. When other users and processes compete for computing resources, placeholder scheduling is still able to keep any real-time load imbalance low (i.e., 2.1%). (Compare Sections 5.1.3 and 5.1.4 and Table 5.1(c),(d).)

Of course, different workloads and different hardware platforms will produce different results. However, the potential of placeholder scheduling as indicated by these initial experiments is encouraging. In fact, the apparent trends in grid and overlay metacomputers play to the strengths of placeholder scheduling. It is likely that many grids or overlay metacomputers will have more than just three systems, many will have a high degree of heterogeneity, and most will be shared among users and jobs. In these situations, dynamic load balancing capabilities will be crucial.

5.2 Experiment #2: Multiple Administrative Domains

The proof-of-concept experiment shows that placeholder scheduling does indeed provide performance benefits to users of an overlay metacomputer while at the same time requiring minimal installed infrastructure. However, the proof-of-concept experiment only demonstrates placeholder scheduling within a single administrative domain. Most overlay metacomputers will cross one or more administrative boundaries, so placeholder scheduling must be validated in a multi-domain environment.

The experiments presented in this section are set up to show placeholder scheduling's ability to handle multiple orthogonal forms of heterogeneity with respect to the sites composing an overlay metacomputer. Once again a sorting application is used because it exhibits well defined behaviour, and is easy to implement and port.

The goals of the sorting experiment are to show the performance of placeholders in four orthogonal dimensions of heterogeneity: (1) parallel vs. sequential computer; (2) machine architecture; (3) distributed vs. shared memory; and (4) administrative domain and local scheduling system. A summary of the systems with respect to these dimensions is shown in Table 5.3. Two placeholders are started on each of System A and System C, and a single placeholder is started on System B. System A and System C use specialized placeholders that are able to dynamically increase or decrease the number of placeholders in the queue depending on measured queue waiting times (as discussed in Section 3.5.4).

An on-line experiment was performed with three different computers, in three different administrative domains, and with three different local schedulers. Each job sorted approximately four million integer keys using four processors on System A, one processor on System B, and eight processors on System C. The workload was identical to that of Section 5.1.3. During the experiment, there were other users running applications at two of the sites (Systems A and C). Although the specific quantitative results from this experiment are not reproducible, the qualitative results are representative. Also, note that System A is administered by the high-performance computing centre of the University of Alberta; System B is in the Department of Computing Science at the University of Alberta, and is effectively under our administrative control; and System C is administered by the University of Calgary.

The throughput, as evidenced by the rate of execution, is shown in Figure 5.10. The cumulative number of work units performed by each system is shown, and the rate of execution is determined by the slopes of the lines. System A exhibits a good initial execution rate, but then suddenly stops executing placeholders. System B, the dedicated sequential machine, exhibits a steady rate of execution. System C is somewhere in between, exhibiting a more or less constant rate of execution, although slower than that of the others. The bottom-most (bar) graph in Figure 5.10 shows the number of work units completed per 5000 second time period.

An interesting point illustrated in Figure 5.10 is the abrupt halt of execution of System A. Upon examination of the PBS logs, it appears as though the placeholders used up the user account's quota of CPU time. As a result, System A becomes unable to execute additional work after roughly 7000 seconds, and this can be perceived as a failure of System A. However, because of the placeholders, the other two systems (Systems B and C) are able to compensate for the loss of System A. After 7000 seconds, only Systems B and C complete work units and are responsible for finishing off the remainder of the workload. Should the loss have occurred without a scheduling system such as placeholder scheduling, users would likely have to discover and correct for this loss on their own (i.e., manual fault tolerance).

Figures 5.11 and 5.12 show the queue lengths and placeholders per queue, respectively. As Figure 5.11 shows, System A is significantly more loaded than System C. However, System A is also more powerful than System C, and therefore execution rates are higher. System A is also able to sustain more placeholders in the queue for the first 7000 seconds, and both queues exhibit

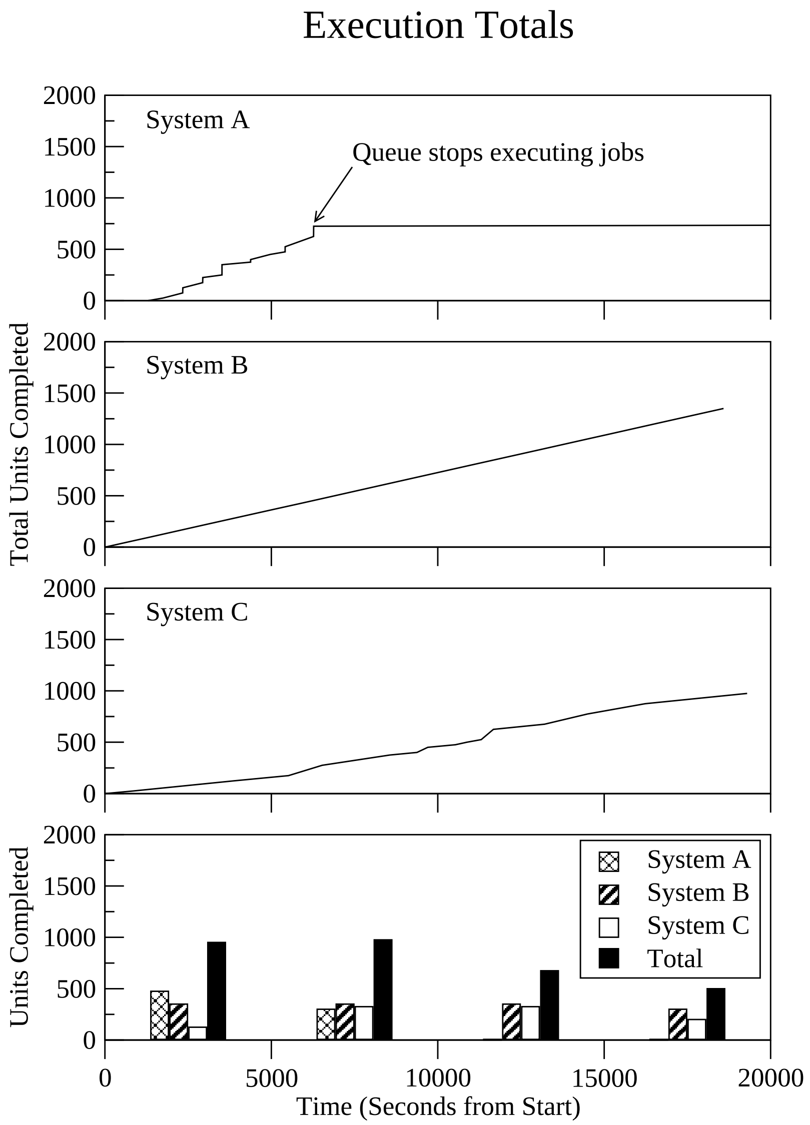


Figure 5.10: Throughput for the Sorting Application

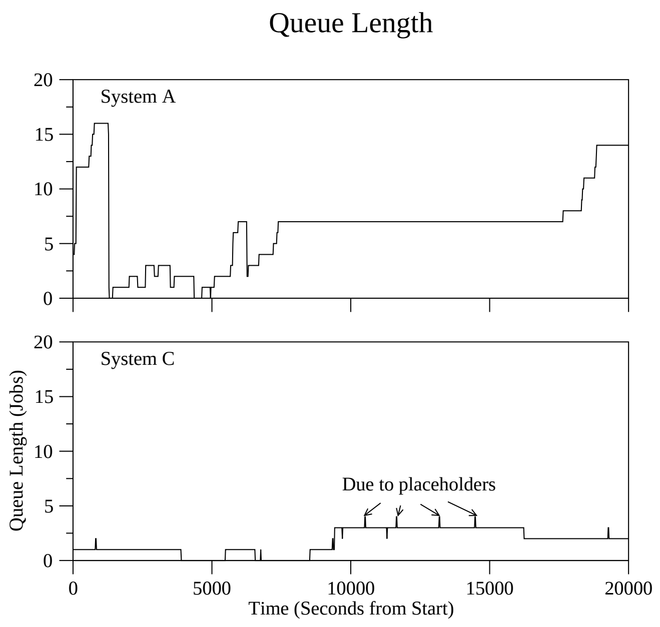


Figure 5.11: Queue Lengths of the Parallel Machines

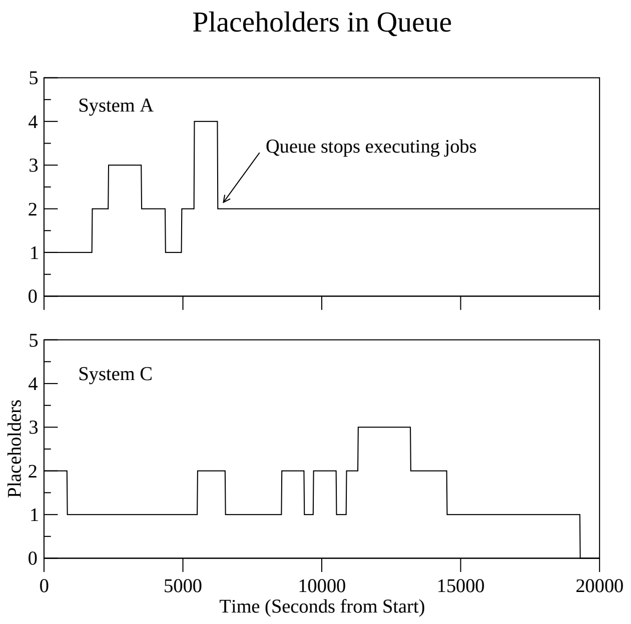


Figure 5.12: Number of Placeholders in Parallel Machine Queues

increases and decreases in placeholder counts due to changing queue conditions (Figure 5.12). It must be emphasized that these results are obtained from computers working on other applications in addition to our own. No attempt has been made to control the queues on Systems A or C.

5.2.1 Summary of Multiple Administrative Domains Experiment

The experiment with multiple administrative domains shows:

1. Placeholder scheduling is capable of scheduling jobs across multiple administrative domains (Table 5.3) while still providing performance benefits (Figure 5.10).

2. Placeholder scheduling is able to dynamically adjust the number of self-regulating placeholders present in a batch scheduled queue according to the observed queuing time (Figure 5.11 and Figure 5.12).

3. Placeholder scheduling is able to compensate for the dynamic failure of a site by avoiding the failed placeholder (System A in Figure 5.10) and concentrating the remainder of the jobs on other, working placeholders (Systems B and C in Figure 5.10).

Placeholder scheduling does not require any special permissions at participating sites to make use of them. By simply using existing infrastructure (SSH), placeholder scheduling is still able to obtain performance benefits for user applications. By incrementally increasing or decreasing the number of placeholders present in local queues, placeholder scheduling can self-adjust to current queue conditions. The loss of System A in this experiment was unintended, but illustrates the truly dynamic nature of a metacomputing environment. Instead of failing the remainder of the workload, placeholder scheduling simply avoided System A and allocated the remainder of the jobs to the other two systems.

5.3 Experiment #3: The CISS Project

The previous experiments show that placeholder scheduling is able to create a metacomputing environment that provides performance benefits to applications across administrative domains. However, the experiments are relatively small (three administrative domains at most). This experiment is meant to show:

1. The scalability of placeholder scheduling with respect to the number of placeholders.

2. The scalability of placeholder scheduling with respect to the number of administrative domains.

3. A real application being computed by placeholder scheduling (as opposed to the trivial sorting application used in Section 5.1 and Section 5.2).

No.	Site	Description	Scheduler	PH
1	athlon-cluster.nic.ualberta.ca	x86 Linux Cluster	PBS	32
2	aurora.nic.ualberta.ca	SGI IRIX	PBS	236
3	brule.cs.ualberta.ca	x86 Linux Cluster	SGE	26
4	bugaboo.hpc.sfu.ca	x86 Linux Cluster	Zero-Infrastructure	192
5	chromosome1.ocgc.ca	SGI Irix	PBS	96
6	deeppurple.sharcnet.ca	Alpha Linux Cluster	Zero-Infrastructure	22
7	driftwood.iam.ubc.ca	x68 Linux Cluster	PBS	16
8	gnome.usask.ca	x86 Linux Cluster	Zero-Infrastructure	32
9	hammerhead.sharcnet.ca	Alpha Linux Cluster	Zero-Infrastructure	22
10	herzberg.physics.mun.ca	SGI Irix	Zero-Infrastructure	20
11	maci-cluster.ucalgary.ca	Tru64 Alpha Cluster	PBS	176
12	mercury.sao.nrc.ca	x86 Linux Cluster	PBS	48
13	minerva.uvic.ca	IBM AIX SP	IBM LoadLeveler	128
14	monolith.uwaterloo.ca	IBM AIX P-Series	Zero-Infrastructure	16
15	myri.ccs.usherbrooke.ca	x86 Linux Cluster	Zero-Infrastructure	8
16	p4-cluster.nic.ualberta.ca	x86 Linux Cluster	PBS	26
17	stokes.clumeq.mcgill.ca	x86 Linux Cluster	PBS	248
18	symphony.unb.ca	IBM AIX SP	IBM LoadLeveler	2
19	white.cs.umanitoba.ca	x86 Linux Cluster	Zero-Infrastructure	22
20	zodiac.chem.ubc.ca	x86 Linux Cluster	PBS	8
			Total	1376

Table 5.4: CISS Sites

The goal of the Canadian Internetworked Scientific Supercomputer (CISS) project [31] is to perform scientific computations on a scale larger than that which can be provided at any single site in Canada. Because of the expansion of the computational aspects of many sciences (physics, biology, chemistry, etc.), the need for additional and more powerful computational resources easily outstrips supply. Also, because of the distribution of computational resources over a wide geographic area, no centralized supercomputing centre currently exists in Canada that can service the largest computational problems.

Thus, there is a need for a metacomputing system to combine some of these distributed resources into a single entity that can be used for computational research. However, because of the large number of administrative domains present across Canada, no metacomputing system currently exists that encompasses a large proportion of the sites. CISS is primarily interested in providing *capacity computing*, in which maximum throughput is obtained for user applications. This contrasts with the goal of *capability computing*, in which a user may use the metacomputer for solving larger and more complicated problems (such as parallel applications). A capacity computing-based system such as CISS provides an excellent situation in which placeholder scheduling can display its benefits.

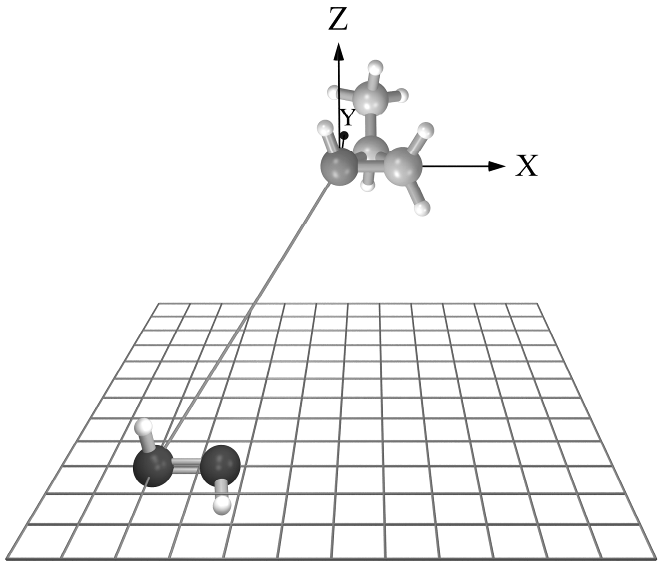


Figure 5.13: Potential Energy Calculations

5.3.1 The CISS-1 Experiment

On November 4, 2002, the CISS-1 experiment combined resources from 20 different high-performance computing systems (16 different institutions, 18 different administrative domains) across Canada to cooperatively execute a computational chemistry parameter space study (Table 5.4). The application calculates the potential energy surface between one of two hydrogen peroxide enantiomers (one of a pair of mirror-image structures for a molecule) and another, fixed molecule (1,2-propyleneimine) (Figure 5.13). Of particular interest are the differences between energy surfaces calculated for the two enantiomers. Dr. Wolfgang Jäger and his group at the University of Alberta [7] planned the experiment and are interpreting the results. From the point of view of placeholder scheduling, the chemistry results are largely uninteresting and so the performance data will instead be examined.

To calculate the potential energy surfaces, a three-dimensional grid was created with the fixed molecule at the centre. Calculations of potential energy between an enantiomer and the fixed molecule were performed at each point of the grid to yield the entire energy surface. Because of the nature of the calculation, the grid may be arbitrarily dense resulting in a potentially infinite number of points. However, the grid was chosen so as to provide adequate granularity for interpreting the potential energy surface while still being computationally feasible (even with a large number of placeholders participating in computing results).

The specific application used in the CISS-1 Experiment is the MOLPRO quantum chemistry package [22]. For this particular experiment, a single MOLPRO invocation requires a site-specific amount of main memory and between 0.5 and 1 GB of temporary disk space. On an average con-

No.	Site	Jobs	Total Time	Mean	σ
1	athlon-cluster.nic.ualberta.ca	307.75	751:12:52	2:26:27	0:46:20
2	aurora.nic.ualberta.ca	437.17	4530:30:47	10:21:46	4:34:15
3	brule.cs.ualberta.ca	433.31	958:13:50	2:12:41	0:33:10
4	bugaboo.hpc.sfu.ca	1575.22	3984:29:37	2:31:46	0:36:55
5	chromosome1.ocgc.ca	136.79	1310:10:33	9:34:41	3:40:41
8	gnome.usask.ca	159.13	729:58:24	4:35:14	1:29:49
10	herzberg.physics.mun.ca	107.37	441:04:44	4:06:29	1:15:01
11	maci-cluster.ucalgary.ca	1064.61	2919:52:07	2:44:34	1:03:15
12	mercury.sao.nrc.ca	397.61	941:16:43	2:22:02	0:43:51
14	monolith.uwaterloo.ca	356.09	383:37:03	1:04:38	0:09:06
15	myri.ccs.usherbrooke.ca	73.02	161:45:13	2:12:54	0:38:28
16	p4-cluster.nic.ualberta.ca	96.56	579:38:53	6:00:10	2:47:13
17	stokes.clumeq.mcgill.ca	2162.47	5625:23:44	2:36:05	0:47:12
19	white.cs.umanitoba.ca	57.46	479:57:20	8:21:10	2:37:03
	other	228.36	1941:46:53	8:30:11	7:38:53
	Overall	7592.94	25738:58:40	3:23:23	2:40:54

Table 5.5: CISS-1 Throughput
All times are reported as H:MM:SS.

temporary processor (such as an AMD Athlon CPU running at 1.3 GHz with 256 KB cache available in the Department of Chemistry at the University of Alberta), MOLPRO takes approximately four hours to compute one data point (provided that the temporary disk space requirement is met).

The CISS-1 experiment was performed as a 24-hour throughput (i.e., capacity computing) test using the MOLPRO application described above. TrellisWeb was used to manage placeholder scheduling because of the convenient user interface it provides. Starting on November 4, 2002 at 12:00 midnight, and ending on November 5, 2002 at 12:00 midnight, placeholders were executed on the systems shown in Table 5.4. The two days prior to November 4, 2002 were used as a ramp-up period to bring all sites up to full capacity, but only the computations performed during the 24-hour period will be discussed here.

The information gathered during the CISS-1 experiment is shown in Table 5.5. A total of 7593 work units (jobs) were completed by CISS-1 in the 24-hour period. The throughput capacity of sites varied greatly, and this is reflected in the number of individual work units each site completed as well as the mean time each site took to complete each one. Most sites exhibit a large amount of variability (as evidenced by σ) due to the fact that MOLPRO requires a variable amount of time depending on the particular input parameters that are being calculated.

Overall, CISS-1 consumed approximately 2.94 years (25738:58:40) of computing time across all sites. The mean time required to compute a work unit was approximately 3.39 hours (3:23:23). However, if we instead assume that a work unit takes approximately four hours to compute on a contemporary x86 cluster processor located in the Department of Chemistry at the University of Alberta, the total computation comes out to about 3.46 years (7292.94 work units at four hours per

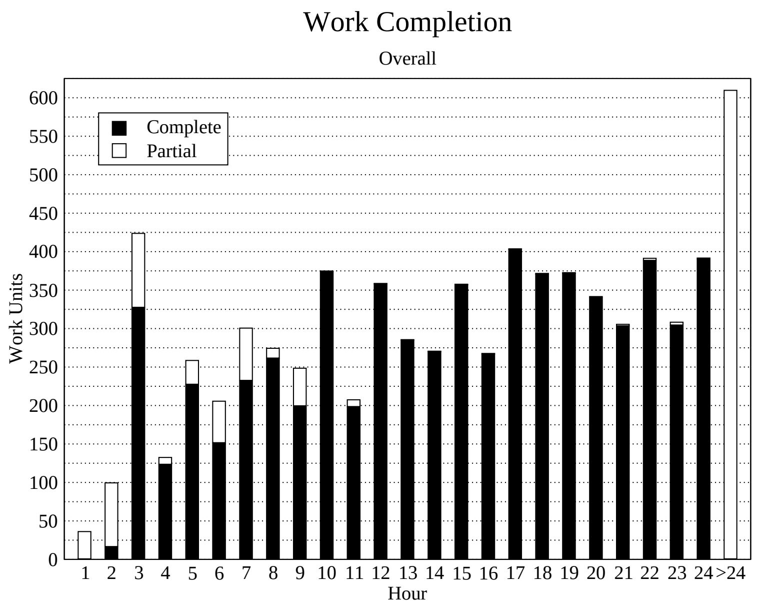


Figure 5.14: Overall Hourly Throughput

work unit).

Performance varies from site to site because of differing numbers of processors, placeholders launched at each site, the availability of local disks, and because of differing capabilities of the various processors (see Table 5.4). The most powerful site in CISS-1, `stokes.clumeq.mcgill.ca`, computed a total of 2162.47 work units with the use of 248 concurrent placeholders. With a mean time of 2:36:05, `stokes.clumeq.mcgill.ca` had a relatively fast processing capability. This stands in contrast to `aurora.nic.ualberta.ca`, which only computed a total of 437.17 work units (20% of `stokes.clumeq.mcgill.ca`) while using 236 concurrent placeholders (95% of `stokes.clumeq.mcgill.ca`). However, work units took a mean time of 10:21:46 to compute at `aurora.nic.ualberta.ca`, resulting in less work being completed there. Differences in computational ability arise from different architectures, different operating systems, and (perhaps most importantly for MOLPRO) different disk access capabilities. MOLPRO performs a great deal of I/O to temporary storage, and so requires quick service times for best performance.

Figure 5.14 and Table 5.6 present the throughput of CISS-1 on an hour-by-hour basis in terms of completed jobs (e.g., a job that finishes at 1:20 p.m. is recorded in hour 13). Partial jobs (jobs that began before and ended after 12:00 midnight on November 4 or jobs that began before and ended after 12:00 midnight on November 5) are included in the figure (grey bars) as proportions of a whole job. CISS-1 reached a maximum throughput at 3 a.m. with 424 jobs completed and is followed closely by 403 jobs completed at 5 p.m. The first two hours represent the continued ramp-up period

Hour	Complete	Partial	Total
1	0	36.19	36.19
2	16	83.47	99.47
3	327	96.77	423.77
4	123	9.51	132.51
5	227	31.62	258.62
6	151	54.58	205.58
7	232	68.62	300.62
8	261	13.39	274.39
9	199	49.52	248.52
10	374	0.71	374.71
11	198	9.38	207.38
12	358	0.00	358.00
13	285	0.00	285.00
14	270	0.00	270.00
15	357	0.00	357.00
16	267	0.00	267.00
17	403	0.00	403.00
18	371	0.00	371.00
19	372	0.00	372.00
20	340	1.65	341.65
21	303	2.49	305.49
22	388	3.35	391.35
23	304	4.29	308.29
24	390	1.73	391.73
$>$ 24	0	609.67	609.67

Table 5.6: Overall Hourly Throughput

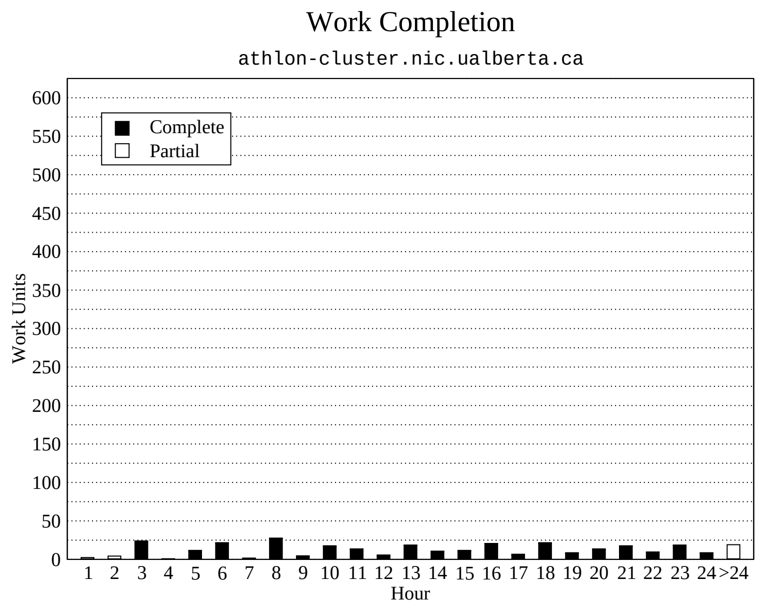


Figure 5.15: `athlon-cluster.nic.ualberta.ca` Hourly Throughput

---

from the two days prior to the 24-hour period. A drop off of throughput occurs at 4 a.m. due to a disk failure at `maci-cluster.ucalgary.ca`. Other sites experienced failures as well, the effects of which are not as noticeable. Subsequently, another ramp-up period brings CISS-1 back up to full speed.

Some of the sites, such as `athlon-cluster.nic.ualberta.ca` (Figure 5.15), `brule.cs.ualberta.ca` (Figure 5.16), and `bugaboo.hpc.sfu.ca` (Figure 5.17), have relatively steady and predictable rates of execution. Although computational ability varies among the sites (the heights of the bars differ), the overall rate within these sites stays fairly consistent. This is largely due to the fact that these sites employed the use of local scratch disks present at each node (all three are clusters). In contrast, sites such as `aurora.nic.ualberta.ca` (Figure 5.18) and, to some extent, `maci-cluster.ucalgary.ca` (Figure 5.19) exhibit spikes in work completion due to the use of a parallel file system (as with `aurora.nic.ualberta.ca`), or some cluster nodes sharing the same scratch disk (as with `maci-cluster.ucalgary.ca`). Because of the temporary storage requirements of MOLPRO, and because such storage is heavily used, MOLPRO jobs may contend with one another in a shared filesystem situation. `aurora.nic.ualberta.ca` experienced a severe I/O bottleneck at the parallel filesystem that resulted in high contention and high mean execution time.

Other sites, such as `p4-cluster.nic.ualberta.ca` (Figure 5.20) and `stokes.clumeq.mcgill.ca` (Figure 5.21), exhibit spikes in completion, but not due to filesystem contention.

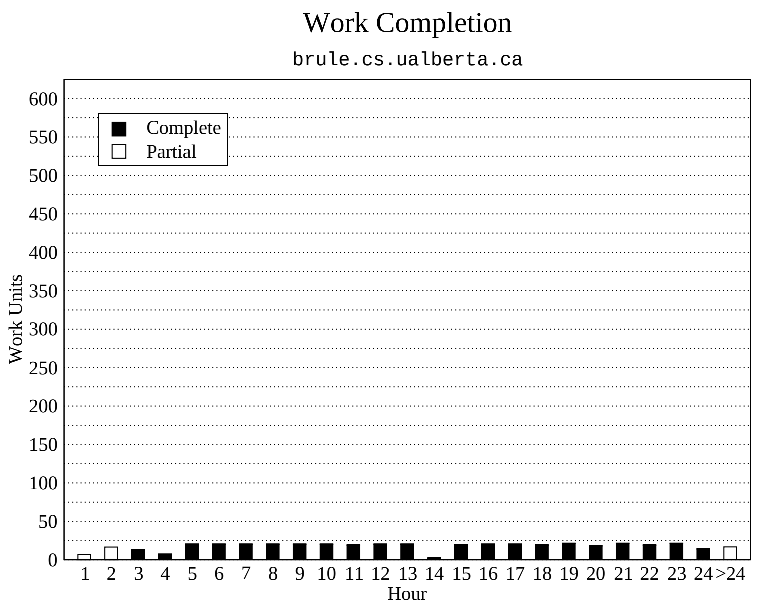


Figure 5.16: `brule.cs.ualberta.ca` Hourly Throughput

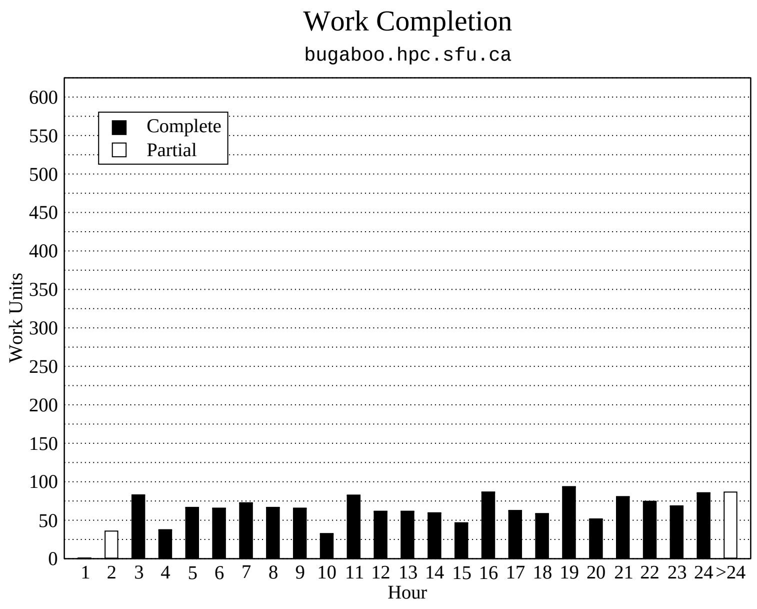


Figure 5.17: `bugaboo.hpc.sfu.ca` Hourly Throughput

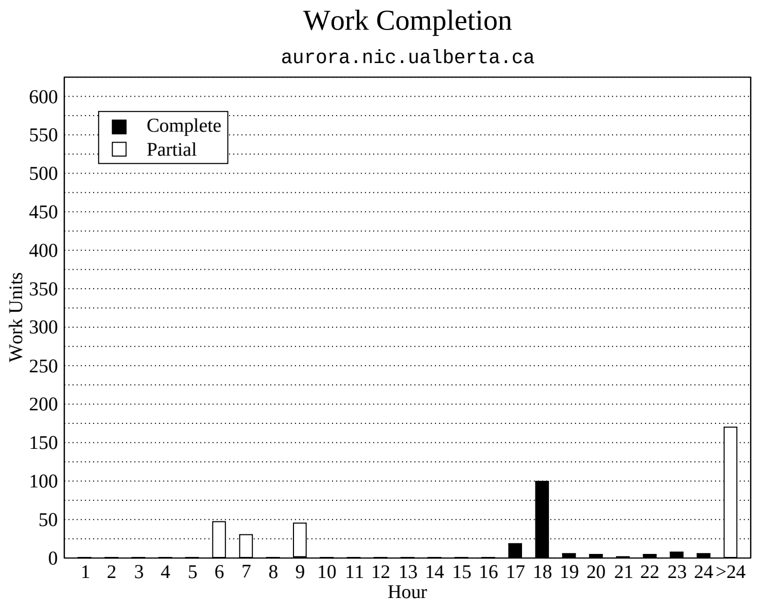


Figure 5.18: `aurora.nic.ualberta.ca` Hourly Throughput

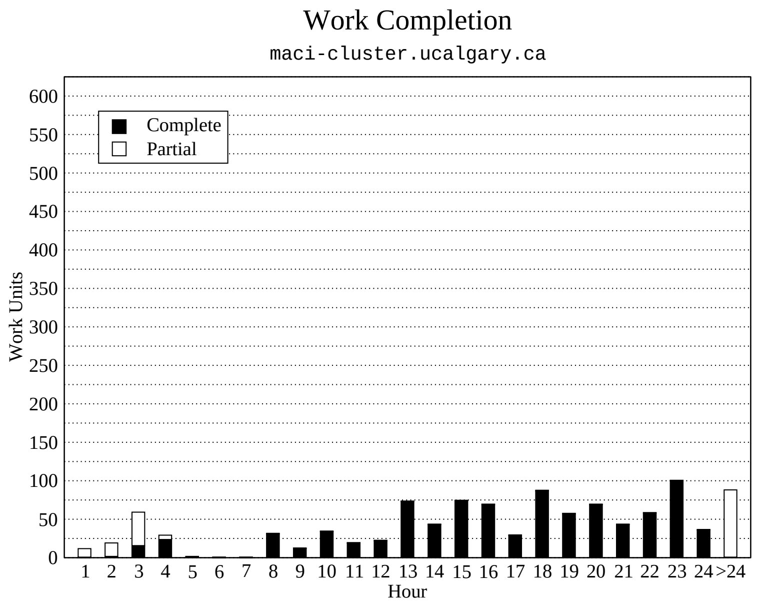


Figure 5.19: `maci-cluster.ucalgary.ca` Hourly Throughput

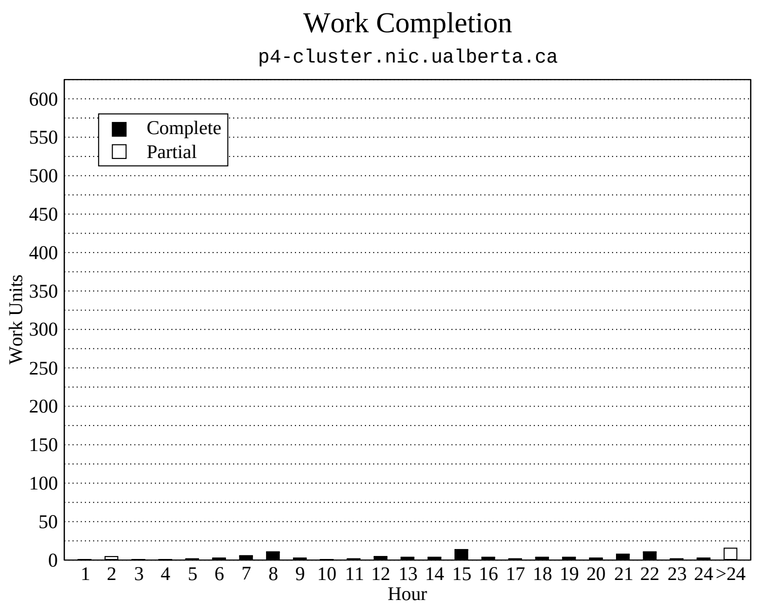


Figure 5.20: `p4-cluster.nic.ualberta.ca` Hourly Throughput

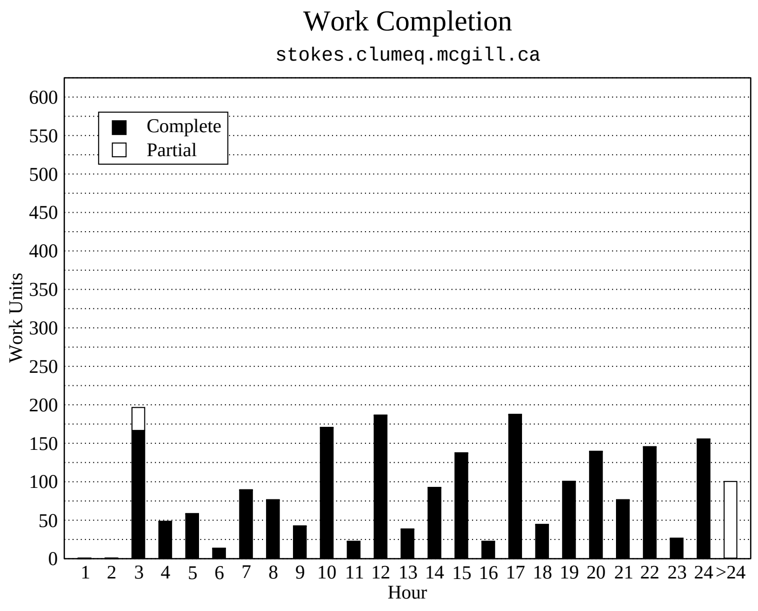


Figure 5.21: `stokes.clumeq.mcgill.ca` Hourly Throughput

Rather, jobs at these sites appear to have clustered around a few key completion times that produce spikes when they overlap. This periodicity of job completion may change over time as the actual run times of MOLPRO jobs can have considerable variance.

The resulting potential energy surface for for $Z = 3$ Å is shown in Figure 5.22. The difference between the two enantiomers is subtle, given the minima and maxima occur at similar XY values. However, the active sites (smaller deviations from zero) are noticeably different between the two. Although it is not possible to physically separate the two H_2O_2 enantiomers, the figure does show that the interaction energy (and therefore stability of the complex) does depend on which enantiomer is involved [14].

5.3.2 Summary of the CISS-1 Experiment

The CISS-1 experiment demonstrates placeholder scheduling's abilities with respect to:

1. **Scalability**: A total of 20 systems contributed to the overall computation. Systems varied in their throughput capacity (Table 5.5), with each system pulling jobs commensurate with their capacity. Placeholder scheduling was able to handle over 1300 placeholders, concurrently executing on 20 different computer systems, for 24 hours.

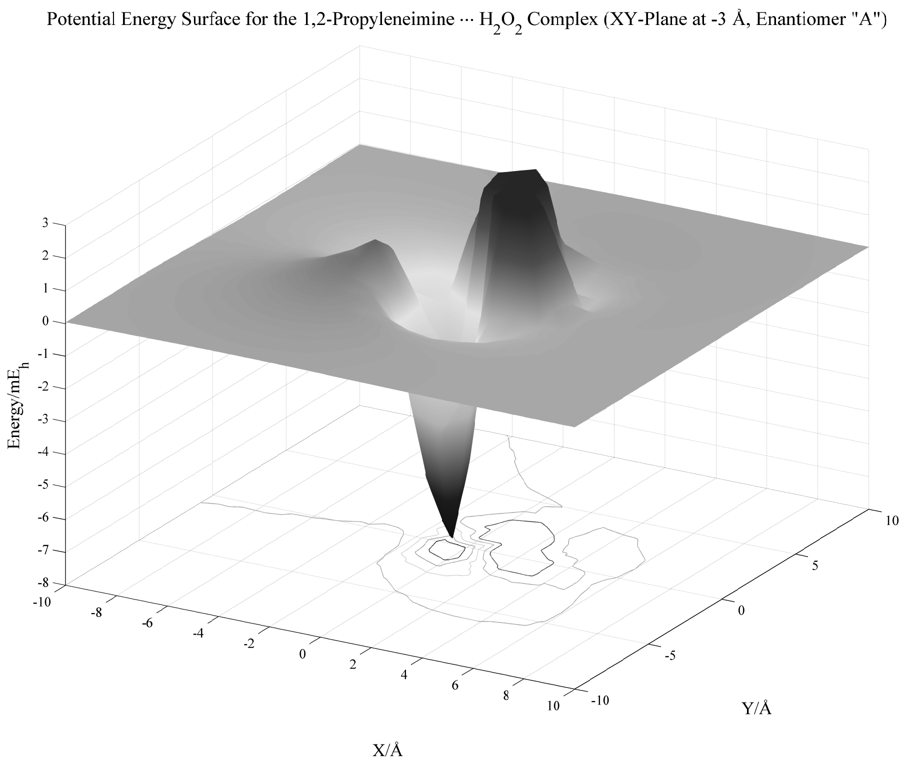
2. **Administrative Domains**: Although shown on a smaller scale in Section 5.2, placeholder scheduling was capable of handling multiple administrative domains. In this experiment, 18 administrative domains existed, each with specific local policies including the choice of batch scheduler (Table 5.4). Placeholder scheduling was able to make use of all of these systems.

3. **Real Application**: Placeholder scheduling executed the MOLPRO quantum chemistry application to produce results of interest to computational chemistry. By using placeholder scheduling, the experiment was able to make use of 2.94 years of CPU time across Canada in a 24-hour period.

5.4 Discussion of Results

The above experiments show placeholder scheduling can be effective in combining the aggregate power of multiple resources for use in a single meaningful computation. This is a significant statement, as many proposed systems never reach the stage of performing significant computations. For example, Condor-G used over 2500 processors at ten sites to compute 540 billion Linear Assignment Problems [11]. Although the number of processors involved in the Condor-G experiment (2500) is larger than that of CISS-1 (1376), more administrative domains (18) and institutions (16) are involved in CISS-1, making it more representative of a diverse metacomputer.

In a controlled environment, such as that of Section 5.1, the performance benefits of placeholder scheduling can easily be observed. Because of the late binding of job to placeholder, placeholder



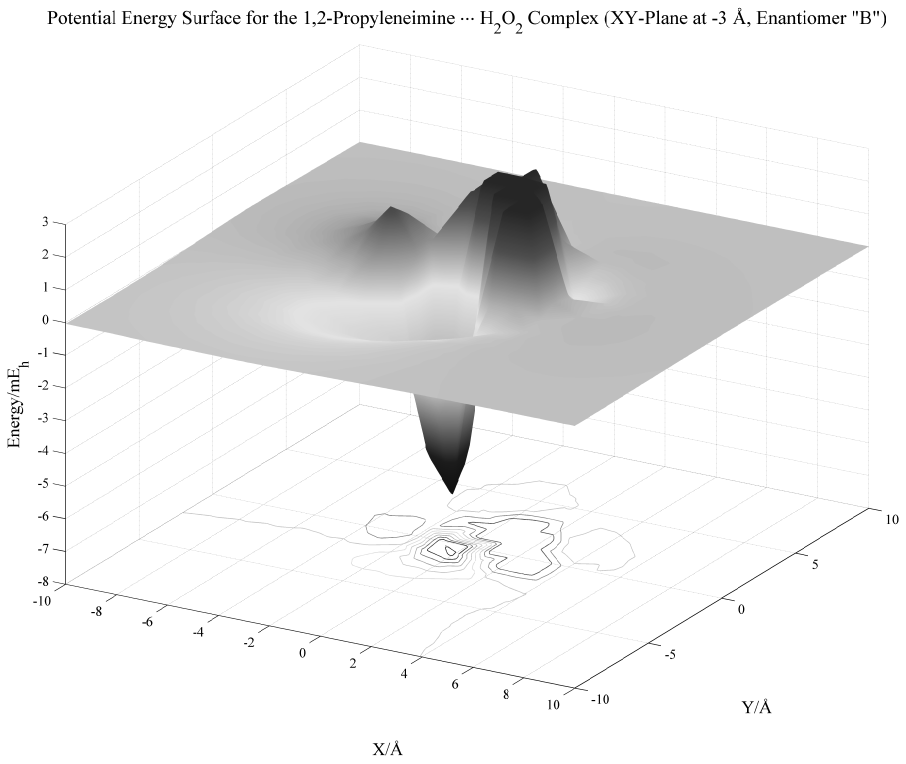


Figure 5.22: Potential Energy Surface for $Z = 3$ Å

scheduling is good at dynamically balancing a load across multiple systems. This is evidenced by the differing capabilities of the systems listed in Table 5.2. Each of the systems are capable of executing a sorting application at a different rate, and placeholder scheduling is able to take this into account when scheduling jobs. Furthermore, the capabilities of the systems may change over time due to the load imposed by other users and applications (Figures 5.2, 5.4, and 5.6) and because of failures (Figure 5.10). Any metacomputing system that is incapable of handling such situations will suffer performance losses when conditions change unexpectedly.

Batch scheduling systems are capable of providing good performance within a single administrative domain. Such schedulers are privy to global knowledge, which makes the task of scheduling that much easier. In general, metacomputers do not have access to the same global information because of the various administrative domains involved and must make do with any information that may be gleaned from the individual systems.

Crossing administrative boundaries causes more problems than just imperfect scheduling information. Infrastructure must be in place that allows jobs to execute on remote machines. Such infrastructure has already been developed (e.g., Globus [8, 9] and Legion [4, 13]), but sometimes requires a single (possibly privileged) account or the use of common software across all sites of a metacomputer. Placeholder scheduling uses existing infrastructure to provide a metacomputer that crosses administrative domains (Section 5.2). Moreover, the performance is not degraded in any way from the situation in which all sites are within the same administrative domain (Section 5.1).

Placeholder scheduling scales from both a relatively small and trivial sorting application (Section 5.1 and Section 5.2) to a Canada-wide scientific experiment (Section 5.3). Because placeholder scheduling relies upon existing commonly-available infrastructure (such as SSH) that has been in place and in use for a number of years, scalability is less of a concern. This comes from the fact that such infrastructure has been extensively tested and refined to discover and correct for deficiencies such as poor scalability.

Chapter 6

Concluding Remarks

Placeholder scheduling within TrellisWeb is effective at achieving the goals set out in Chapter 1. The mechanism of placeholder scheduling is deceptively simple in concept, yet highly flexible in practice, and this makes it an excellent light-weight (in terms of required infrastructure) metacomputing system. Although the scheduling policy currently implemented by placeholder scheduling is simple (FCFS), the potential exists for much more complex and interesting policies (such as dependency-based scheduling). Placeholder scheduling currently concentrates on the mechanism of global metacomputer scheduling.

Placeholder scheduling exists at a higher level of abstraction than other forms of batch scheduling. Most batch scheduling systems exist primarily to control access to computational resources within a single site. Some modern batch scheduling systems (e.g., LSF) are able to control multiple sites simultaneously, but all such sites must be using the same software. As experiments such as CISS-1 show, assuming a homogeneous scheduling environment is generally not possible. As a result, placeholder scheduling layers on top of existing scheduling systems by making use of the ubiquitous notion of a job script. Such scripts exist for many systems, and placeholders are simply implemented in the form required for the particular system (as for PBS in Figure 3.4, for example). By layering on top of existing scheduling systems, placeholder scheduling allows local policies to be enforced and allows local users to use the system without placeholder scheduling.

Computing sites may be very diverse due to certain parameters selected by local administration. This results in a great deal of heterogeneity present in a multi-site metacomputer, the most important aspect of which is the mix of different administrative domains. Different administrative domains may address similar problems in different ways, and negotiating agreements among them can be difficult and time-consuming. Placeholder scheduling avoids such normalization of administrative domains by minimizing the amount of infrastructure required at the sites and adhering to local policies within the placeholder scripts. For placeholder scheduling to make use of a site, all that is required is a normal user account and SSH access to it. This is the same requirement for interactive remote access, and so it not significantly different than that required by a normal remote user. Likewise, placeholder scripts can be written in such a way that they adhere to local policies such as the local batch scheduler. Because placeholder scheduling assumes such little infrastructure, users do not need special permission to use it at sites of their choosing.

Placeholder scheduling must be able to provide users with noticeable and meaningful performance benefits to be considered useful. The goal of the user in using the additional resources provided by a metacomputing environment is to decrease the time it takes for them to complete their work, and so this metric is used to evaluate the performance of placeholder scheduling. To minimize the amount of time required to complete a set of jobs, placeholder scheduling load balances the jobs across all available placeholders. Because of the pull model and late binding, each placeholder pulls work at a rate equal to that of its computational ability. Without the assistance of placeholder scheduling, forecasting the computational ability *a priori* may lead to poor decisions of where to place jobs due to the dynamic nature of sites involved in an overlay metacomputer (Section 5.1.1, for example). The dynamic binding of placeholder to job is able to account for fluctuations in placeholder performance, resulting in much better load balancing and reduced makespans for the user.

The primary focus of the CISS project (using placeholder scheduling and TrellisWeb) is on scientific computation. The application used in Section 5.3.1 is embarrassingly parallel, and is of little interest in the area of parallel computing. However, embarrassingly parallel applications (such as MOLPRO, as used with the CISS-1 experiment) represent an important aspect of computational science, and are therefore highly relevant to high-performance computing in general. It should be noted that although placeholder scheduling is not expressly restricted to executing embarrassingly parallel applications, it does not provide the ability to concurrently schedule a single job across multiple administrative domains. This is because most parallel applications (other than embarrassingly parallel ones) involve a great deal of communication between processes, communication which would be relatively slow across wide area links between some of the subprocesses. Instead, placeholder scheduling allows for the placement of a parallel job at a single site so that communication is as efficient as possible. To put this another way, placeholder scheduling is interested in capacity computing rather than capability computing. Placeholder scheduling tries to get as much done within as little time as possible rather than trying to provide maximum functionality to the user.

Placeholder scheduling, in one form or another, has appeared in the past. Systems such as SETI@Home [28] employ clients that are essentially special-purpose placeholders. Work is retrieved from a server on demand (pull model) and computed offline while the computer is idle. The contribution of this work is that placeholders are general entities capable of computing any requested job and they are layered on top of existing batch schedulers and security infrastructure. The concept of placeholders and how they work is of primary importance, with implementations being developed and described so that experiments can be performed and results obtained. Although placeholders are dependent on batch schedulers for the most part (except for zero-infrastructure placeholders), batch schedulers are unlikely to change significantly in the near future because of their roles as resource arbitrators at high-performance computing sites. Therefore, it is hoped that the legacy of this work will be the concept of placeholder scheduling and its simplicity of use for metacomputing applications.

6.1 Future Work

As was stated previously, placeholder scheduling focusses primarily on the mechanisms for global metacomputer scheduling and largely ignores complex policies. Most modern scheduling environments (such as PBS) are capable of using far more sophisticated scheduling policies than simple FCFS. Features such as run time prediction, backfilling, and preemption may be used to generate schedules far superior to that of a simple FCFS schedule. Because most of the scheduling logic exists within the command-line server, modifications to the command-line server can affect the scheduling policy. Situations such as when one user has (static or dynamic) priority over another user can be implemented in the logic of the command-line server, and represent a more complex policy than FCFS. Likewise, a command-line server could conceivably build a history of job executions and corresponding waiting and executing times so that an estimate could be generated for future runs. At any rate, policies more diverse and interesting than FCFS are within the realm of possibilities for placeholder scheduling.

The experiments of Chapter 5 show that placeholder scheduling is able to successfully schedule a scientifically-significant application across a number of sites and administrative domains. What remains to be shown is the ability for placeholder scheduling to perform this feat for multiple concurrent applications. In the future, under the banner of the CISS project, multiple such applications similar to that of Section 5.3.1 may be executed concurrently with placeholder scheduling and TrellisWeb. Such an experiment would serve to show that placeholder scheduling is scalable with respect to the number of applications, in addition to being scalable with respect to sites and administrative domains.

Placeholder scheduling is easy to deploy and use because it does not require sophisticated underlying infrastructure beyond what already exists at most sites. However, this does not mean that placeholder scheduling is fundamentally unable to take advantage of such infrastructure. Some collection of sites may have agreed to form a Globus grid, and placeholder scheduling should be able to take advantage of such a grid. All that would be required is for placeholders and TrellisWeb to interact with Globus software instead of batch scheduling software. By creating a meta-metacomputer out of underlying computational grids, placeholder scheduling could provide the variety of Every-Ware [34] while still adhering to the metaqueue model of scheduling. Conversely, a Globus GRAM [5] could be developed to use placeholder scheduling. Either way, placeholder scheduling could work with or without grid software in place.

Bibliography

[1] D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 520–528, Cancun, Mexico, May 2000.

[2] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: A Tool for Performing Parametised Simulations using Distributed Workstations. In *Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing*, August 1995.

[3] D. J. Barrett and R. E. Silverman. *SSH, the Secure Shell: The Definitive Guide*. O'Reilly and Associates, Sebastopol, CA, 2001.

[4] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. Resource Management in Legion. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1659 of *Lecture Notes in Computer Science*, pages 162–178. Springer Verlag, 1999.

[5] K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 62–82. Springer Verlag, 1998.

[6] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and Practice in Parallel Job Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. Springer Verlag, 1997.

[7] Laboratory for the Study of Intermolecular Interactions. http://www.chem.ualberta.ca/~jaeger/.

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

[9] I. Foster and C. Kesselman. The Globus Project: A Status Report. In *Proceedings of the 1998 IPPS/SPDP Heterogeneous Computing Workshop*, pages 4–18, 1998.

[10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[11] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing*, August 2001.

[12] M. Goldenberg. A System For Structured DAG Scheduling. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, in preparation.

[13] A. S. Grimshaw and W. A. Wulf. Legion – A View From 50,000 Feet. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, U.S.A., August 1996. IEEE Computer Society Press.

[14] W. Jäger and A. Huckauf. Private Email Correspondence.

[15] N. H. Kapadia and J. A. B. Fortes. PUNCH: An Architecture for Web-Enabled Wide-Area Network Computing. *Cluster Computing*, 2:153–164, 1999.

[16] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19(10):1079–1103, October 1993.

[17] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of the 1988 IEEE Eighth International Conference on Distributed Computing Systems*, 1988.

[18] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), June 1997.

[19] Load Sharing Facility (LSF). http://www.platform.com/.

[20] G. Ma and P. Lu. PBSWeb: A Web-based Interface to the Portable Batch System. In *Proceedings of the 12th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 24–30, Las Vegas, Nevada, U.S.A., November 2000.

[21] G. Ma, V. Salamon, and P. Lu. Security and History Management Improvements to PBSWeb. In *Proceedings of the 15th International Symposium on High Performance Computing Systems and Applications*, Windsor, Ontario, Canada, June 2001.

[22] MOLPRO Quantum Chemistry Package. http://www.molpro.net/.

[23] A. Natrajan, A. Nguyen-Tuong, M. A. Humphrey, and A. S. Grimshaw. The Legion Grid Portal. Available at http://legion.virginia.edu/papers.html.

[24] C. Pinchak, P. Lu, and M. Goldenberg. Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. In D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 202–225. Springer Verlag, 2002.

[25] Portable Batch System (PBS). http://www.openpbs.org/.

[26] J. Pruyne and M. Livny. Interfacing Condor and PVM to Harness the Cycles of Workstation Clusters. *Future Generation Computer Systems*, 12(1):67–85, May 1996.

[27] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation In Parallel Machines. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 237–245, Hilton Head, South Carolina, U.S.A., July 1991. ACM Press.

[28] SETI@Home. http://setiathome.ssl.berkeley.edu/.

[29] J. Siegel and P. Lu. User-Level Remote Data Access in Overlay Metacomputers. In *Proceedings of the 4th IEEE International Conference on Cluster Computing*, pages 480–483, Chicago, Illinois, U.S.A., September 2002.

[30] Sun Grid Engine (SGE). http://www.sun.com/software/gridware/sge.html.

[31] The Canadian Internetworked Scientific Supercomputer Project. http://www.cs.ualberta.ca/˜ciss/.

[32] The Trellis Project. http://www.cs.ualberta.ca/˜paullu/Trellis/.

[33] M. Thomas, S. Mock, M. Dahan, K. Mueller, D. Sutton, and J. R. Boisseau. The GridPort Toolkit: a System for Building Grid Portals. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing*, August 2001.

[34] R. Wolski, J. Brevik, C. Krintz, G. Obertelli, N. Spring, and A. Su. Running EveryWare on the Computational Grid. In *Proceedings of Supercomputing '99*, Portland, Oregon, U.S.A., November 1999.

Appendix A

Individual CISS-1 Site Results

The results for individual sites of the CISS-1 experiment, described in Section 5.3.1, are included here. Some have been omitted at the request of site administration, and so appear in the combined "Other" category (Figure A.15 and Table A.15). Some figures are repeated here for the sake of completeness.

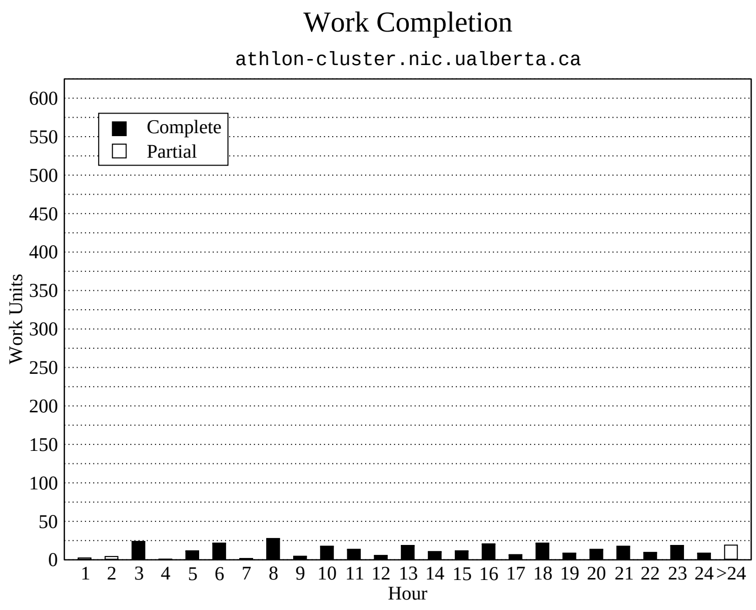


Figure A.1: `athlon-cluster.nic.ualberta.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	2.42	2.42
2	0	4.25	4.25
3	23	0.96	23.96
4	0	0.00	0.00
5	11	0.00	11.00
6	21	0.00	21.00
7	1	0.00	1.00
8	27	0.00	27.00
9	4	0.00	4.00
10	17	0.00	17.00
11	13	0.00	13.00
12	5	0.00	5.00
13	18	0.00	18.00
14	10	0.00	10.00
15	11	0.00	11.00
16	20	0.00	20.00
17	6	0.00	6.00
18	21	0.00	21.00
19	8	0.00	8.00
20	13	0.00	13.00
21	17	0.00	17.00
22	9	0.00	9.00
23	18	0.00	18.00
24	8	0.00	8.00
>24	0	19.12	19.12

Table A.1: `athlon-cluster.nic.ualberta.ca` Hourly Throughput

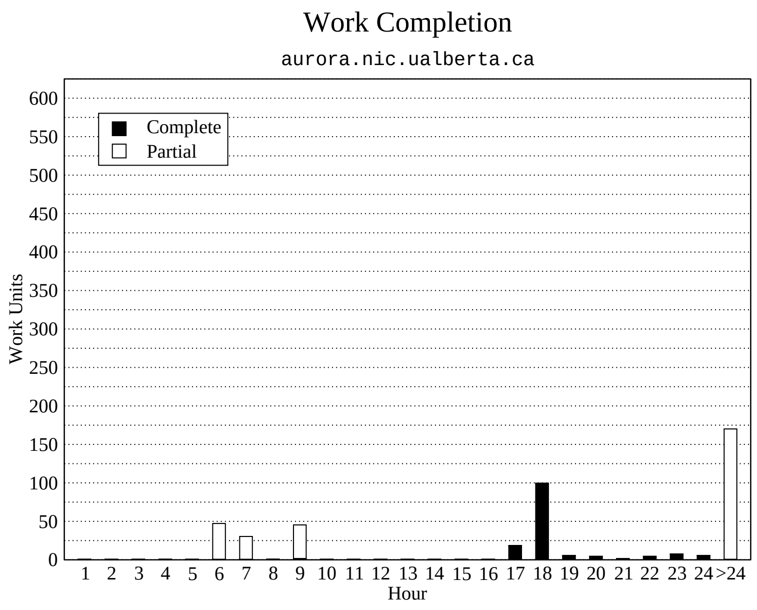


Figure A.2: `aurora.nic.ualberta.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.00	0.00
2	0	0.00	0.00
3	0	0.00	0.00
4	0	0.53	0.53
5	0	0.00	0.00
6	0	47.21	47.21
7	0	30.33	30.33
8	0	0.64	0.64
9	1	44.37	45.37
10	0	0.00	0.00
11	0	0.00	0.00
12	0	0.00	0.00
13	0	0.00	0.00
14	0	0.00	0.00
15	0	0.00	0.00
16	0	0.00	0.00
17	18	0.00	18.00
18	99	0.00	99.00
19	5	0.00	5.00
20	4	0.00	4.00
21	1	0.00	1.00
22	4	0.00	4.00
23	7	0.00	7.00
24	5	0.00	5.00
> 24	0	170.11	170.11

Table A.2: `aurora.nic.ualberta.ca` Hourly Throughput

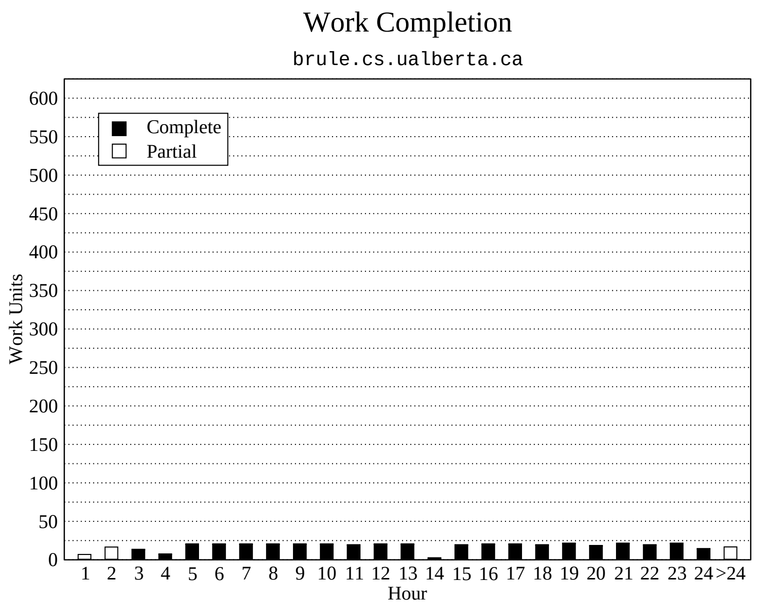


Figure A.3: `brule.cs.ualberta.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	6.94	6.94
2	0	16.62	16.62
3	13	0.00	13.00
4	7	0.00	7.00
5	20	0.00	20.00
6	20	0.00	20.00
7	20	0.00	20.00
8	20	0.00	20.00
9	20	0.00	20.00
10	20	0.00	20.00
11	19	0.00	19.00
12	20	0.00	20.00
13	20	0.00	20.00
14	2	0.00	2.00
15	19	0.00	19.00
16	20	0.00	20.00
17	20	0.00	20.00
18	19	0.00	19.00
19	21	0.00	21.00
20	18	0.00	18.00
21	21	0.00	21.00
22	19	0.00	19.00
23	21	0.00	21.00
24	14	0.00	14.00
> 24	0	16.75	16.75

Table A.3: `brule.cs.ualberta.ca` Hourly Throughput

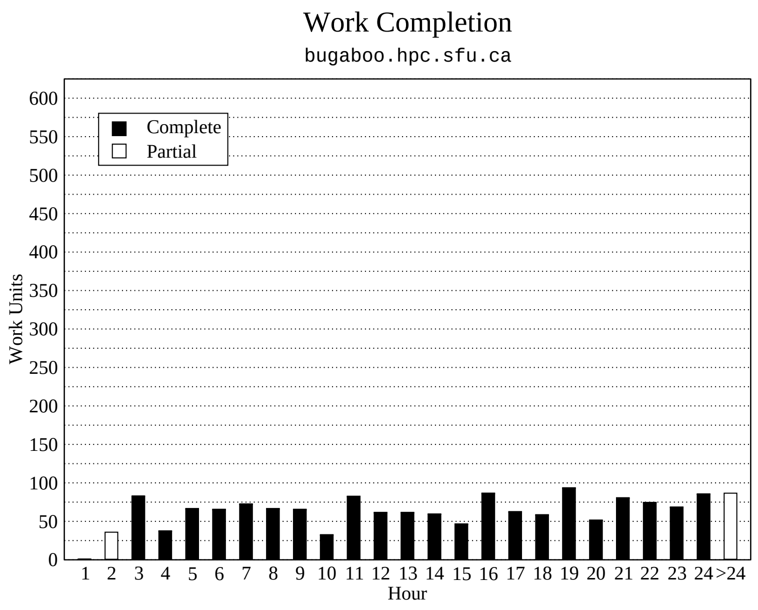


Figure A.4: `bugaboo.hpc.sfu.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.84	0.84
2	0	35.88	35.88
3	82	0.92	82.92
4	37	0.00	37.00
5	66	0.00	66.00
6	65	0.00	65.00
7	72	0.00	72.00
8	66	0.00	66.00
9	65	0.00	65.00
10	32	0.00	32.00
11	82	0.00	82.00
12	61	0.00	61.00
13	61	0.00	61.00
14	59	0.00	59.00
15	46	0.00	46.00
16	86	0.00	86.00
17	62	0.00	62.00
18	58	0.00	58.00
19	93	0.00	93.00
20	51	0.00	51.00
21	80	0.00	80.00
22	74	0.00	74.00
23	68	0.00	68.00
24	85	0.00	85.00
> 24	0	86.58	86.58

Table A.4: `bugaboo.hpc.sfu.ca` Hourly Throughput

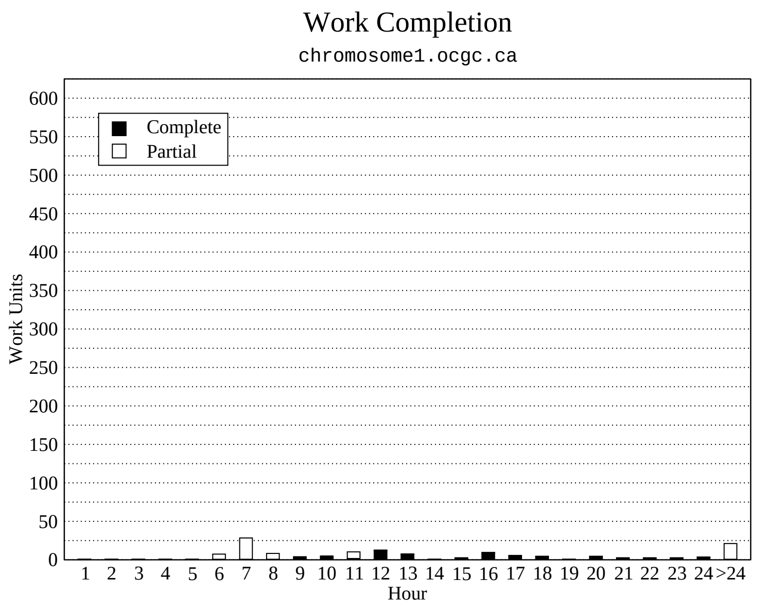


Figure A.5: `chromosome1.ocgc.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.00	0.00
2	0	0.00	0.00
3	0	0.00	0.00
4	0	0.47	0.47
5	0	0.00	0.00
6	0	7.37	7.37
7	0	28.38	28.38
8	0	8.30	8.30
9	2	2.03	4.03
10	4	0.71	4.71
11	1	9.38	10.38
12	12	0.00	12.00
13	7	0.00	7.00
14	0	0.00	0.00
15	2	0.00	2.00
16	9	0.00	9.00
17	5	0.00	5.00
18	4	0.00	4.00
19	0	0.00	0.00
20	4	0.00	4.00
21	2	0.00	2.00
22	2	0.00	2.00
23	2	0.00	2.00
24	3	0.00	3.00
> 24	0	21.15	21.15

Table A.5: `chromosome1.ocgc.ca` Hourly Throughput

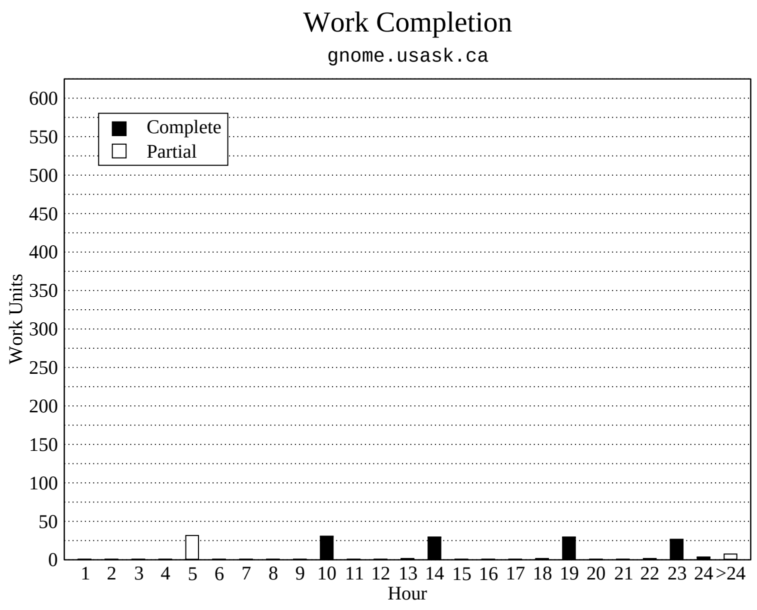


Figure A.6: `gnome.usask.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.00	0.00
2	0	0.00	0.00
3	0	0.00	0.00
4	0	0.00	0.00
5	0	31.62	31.62
6	0	0.00	0.00
7	0	0.00	0.00
8	0	0.00	0.00
9	0	0.00	0.00
10	30	0.00	30.00
11	0	0.00	0.00
12	0	0.00	0.00
13	1	0.00	1.00
14	29	0.00	29.00
15	0	0.00	0.00
16	0	0.00	0.00
17	0	0.00	0.00
18	1	0.00	1.00
19	29	0.00	29.00
20	0	0.00	0.00
21	0	0.00	0.00
22	1	0.00	1.00
23	26	0.00	26.00
24	3	0.00	3.00
> 24	0	7.51	7.51

Table A.6: `gnome.usask.ca` Hourly Throughput

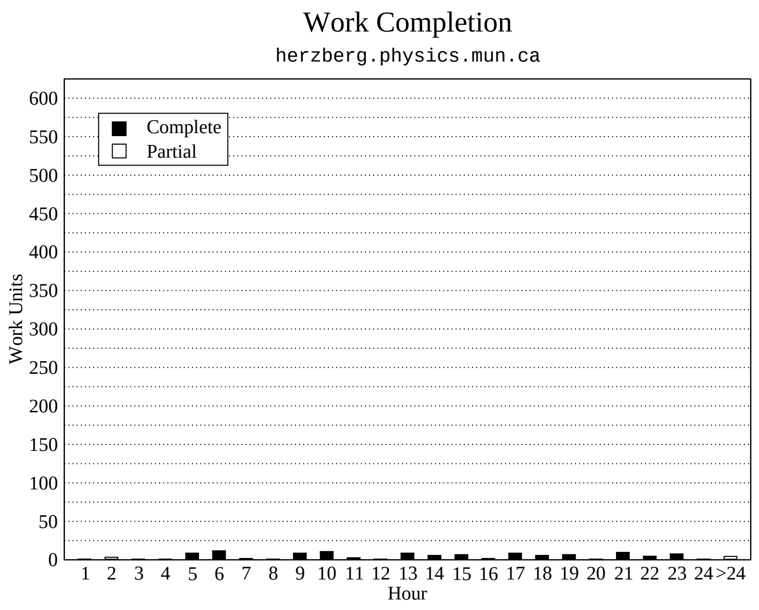


Figure A.7: `herzberg.physics.mun.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.52	0.52
2	0	3.34	3.34
3	0	0.00	0.00
4	0	0.00	0.00
5	8	0.00	8.00
6	11	0.00	11.00
7	1	0.00	1.00
8	0	0.00	0.00
9	8	0.00	8.00
10	10	0.00	10.00
11	2	0.00	2.00
12	0	0.00	0.00
13	8	0.00	8.00
14	5	0.00	5.00
15	6	0.00	6.00
16	1	0.00	1.00
17	8	0.00	8.00
18	5	0.00	5.00
19	6	0.00	6.00
20	0	0.00	0.00
21	9	0.00	9.00
22	4	0.00	4.00
23	7	0.00	7.00
24	0	0.00	0.00
>24	0	4.50	4.50

Table A.7: `herzberg.physics.mun.ca` Hourly Throughput

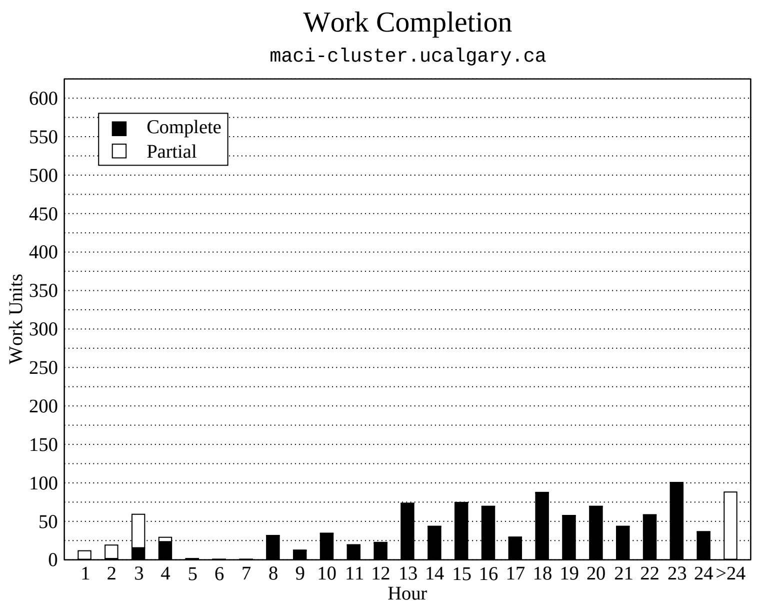


Figure A.8: `maci-cluster.ucalgary.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	11.76	11.76
2	1	18.24	19.24
3	15	44.17	59.17
4	23	6.28	29.28
5	1	0.00	1.00
6	0	0.00	0.00
7	0	0.00	0.00
8	31	0.00	31.00
9	12	0.00	12.00
10	34	0.00	34.00
11	19	0.00	19.00
12	22	0.00	22.00
13	73	0.00	73.00
14	44	0.00	44.00
15	74	0.00	74.00
16	69	0.00	69.00
17	29	0.00	29.00
18	87	0.00	87.00
19	57	0.00	57.00
20	69	0.00	69.00
21	43	0.00	43.00
22	58	0.00	58.00
23	100	0.00	100.00
24	36	0.00	36.00
> 24	0	88.16	88.16

Table A.8: `maci-cluster.ucalgary.ca` Hourly Throughput

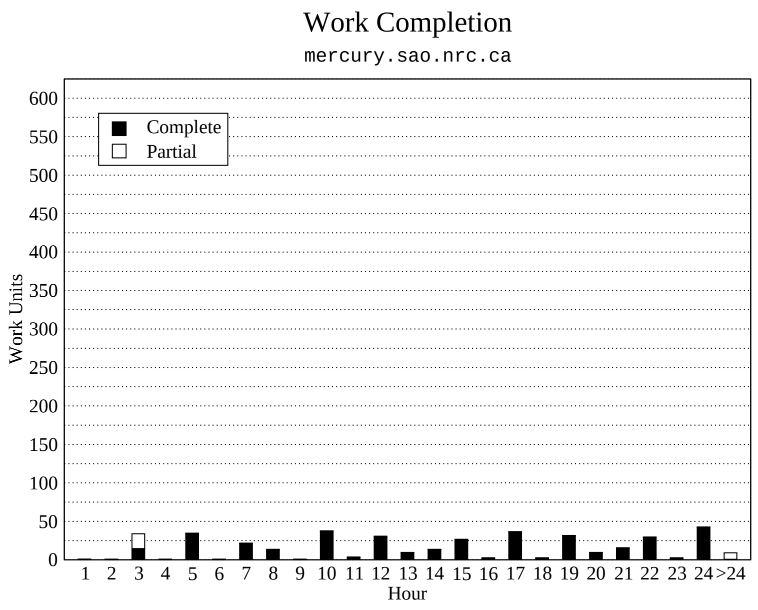


Figure A.9: `mercury.sao.nrc.ca` Hourly Throughput

Hour	**Complete**	**Partial**	**Total**
1	0	0.66	0.66
2	0	0.00	0.00
3	14	19.84	33.84
4	0	0.00	0.00
5	34	0.00	34.00
6	0	0.00	0.00
7	21	0.00	21.00
8	13	0.00	13.00
9	0	0.00	0.00
10	37	0.00	37.00
11	3	0.00	3.00
12	30	0.00	30.00
13	9	0.00	9.00
14	13	0.00	13.00
15	26	0.00	26.00
16	2	0.00	2.00
17	36	0.00	36.00
18	2	0.00	2.00
19	31	0.00	31.00
20	9	0.00	9.00
21	15	0.00	15.00
22	29	0.00	29.00
23	2	0.00	2.00
24	42	0.00	42.00
>24	0	9.12	9.12

Table A.9: `mercury.sao.nrc.ca` Hourly Throughput

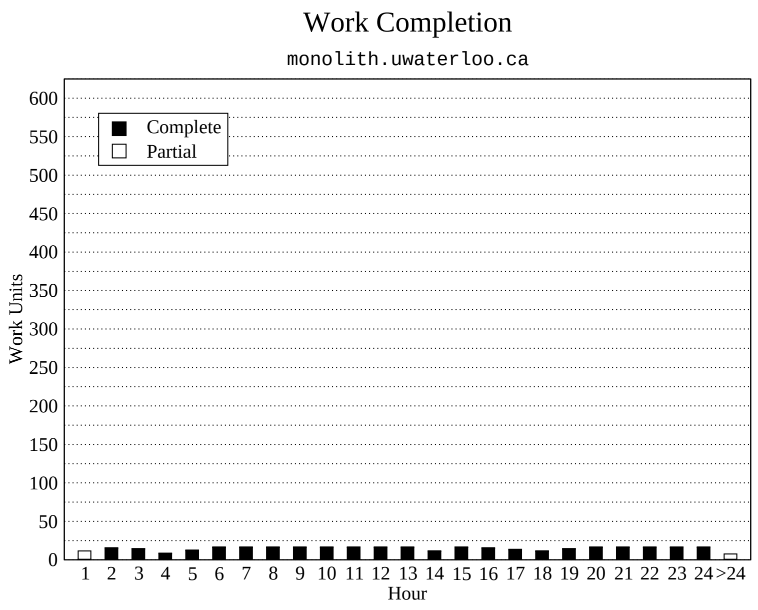


Figure A.10: `monolith.uwaterloo.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	11.52	11.52
2	15	0.00	15.00
3	14	0.00	14.00
4	8	0.00	8.00
5	12	0.00	12.00
6	16	0.00	16.00
7	16	0.00	16.00
8	16	0.00	16.00
9	16	0.00	16.00
10	16	0.00	16.00
11	16	0.00	16.00
12	16	0.00	16.00
13	16	0.00	16.00
14	11	0.00	11.00
15	16	0.00	16.00
16	15	0.00	15.00
17	13	0.00	13.00
18	11	0.00	11.00
19	14	0.00	14.00
20	16	0.00	16.00
21	16	0.00	16.00
22	16	0.00	16.00
23	16	0.00	16.00
24	16	0.00	16.00
> 24	0	7.57	7.57

Table A.10: `monolith.uwaterloo.ca` Hourly Throughput

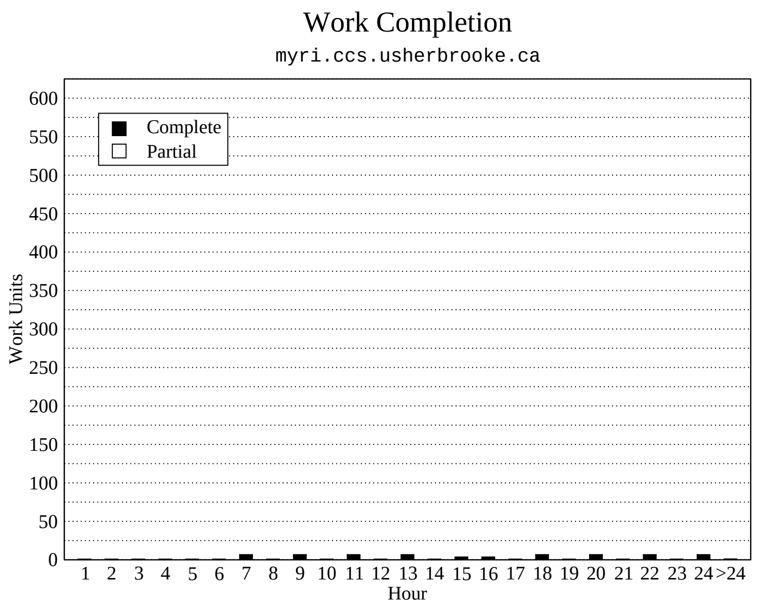


Figure A.11: `myri.ccs.usherbrooke.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.00	0.00
2	0	0.00	0.00
3	0	0.00	0.00
4	0	0.00	0.00
5	0	0.00	0.00
6	0	0.00	0.00
7	6	0.00	6.00
8	0	0.00	0.00
9	6	0.00	6.00
10	0	0.00	0.00
11	6	0.00	6.00
12	0	0.00	0.00
13	6	0.00	6.00
14	0	0.00	0.00
15	3	0.00	3.00
16	3	0.00	3.00
17	0	0.00	0.00
18	6	0.00	6.00
19	0	0.00	0.00
20	6	0.00	6.00
21	0	0.00	0.00
22	6	0.00	6.00
23	0	0.00	0.00
24	6	0.00	6.00
> 24	0	1.02	1.02

Table A.11: `myri.ccs.usherbrooke.ca` Hourly Throughput

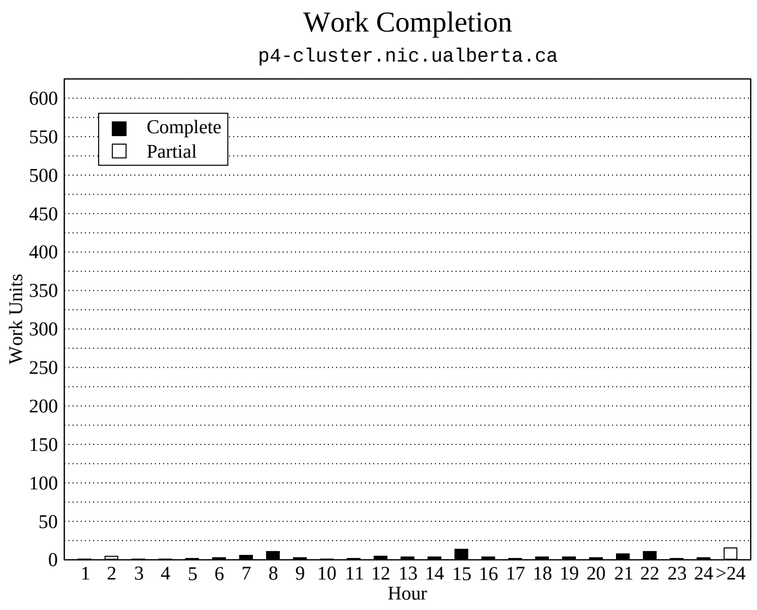


Figure A.12: `p4-cluster.nic.ualberta.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.23	0.23
2	0	4.57	4.57
3	0	0.35	0.35
4	0	0.00	0.00
5	1	0.00	1.00
6	2	0.00	2.00
7	5	0.00	5.00
8	10	0.00	10.00
9	2	0.00	2.00
10	0	0.00	0.00
11	1	0.00	1.00
12	4	0.00	4.00
13	3	0.00	3.00
14	3	0.00	3.00
15	13	0.00	13.00
16	3	0.00	3.00
17	1	0.00	1.00
18	3	0.00	3.00
19	3	0.00	3.00
20	2	0.00	2.00
21	7	0.00	7.00
22	10	0.00	10.00
23	1	0.00	1.00
24	2	0.00	2.00
> 24	0	15.42	15.42

Table A.12: `p4-cluster.nic.ualberta.ca` Hourly Throughput

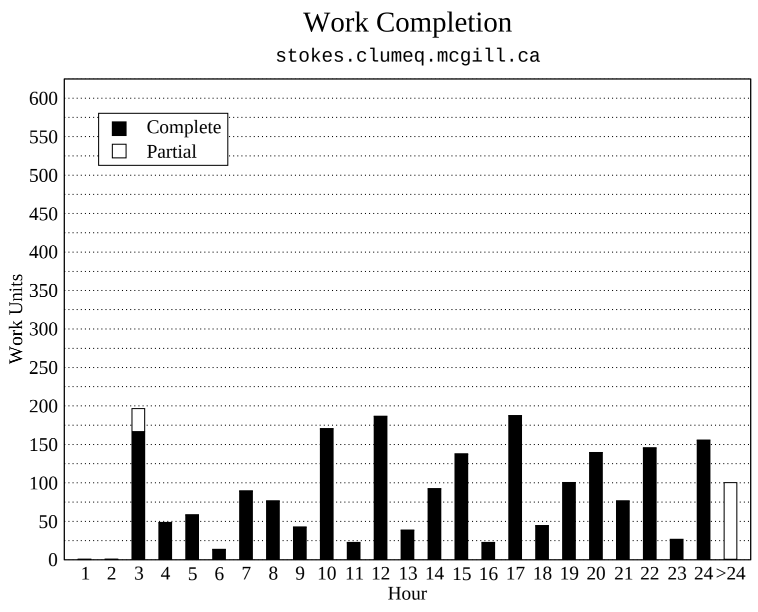


Figure A.13: `stokes.clumeq.mcgill.ca` Hourly Throughput

Hour	Complete	Partial	Total
1	0	0.00	0.00
2	0	0.56	0.56
3	166	30.54	196.54
4	48	0.00	48.00
5	58	0.00	58.00
6	13	0.00	13.00
7	89	0.00	89.00
8	76	0.00	76.00
9	42	0.00	42.00
10	170	0.00	170.00
11	22	0.00	22.00
12	186	0.00	186.00
13	38	0.00	38.00
14	92	0.00	92.00
15	137	0.00	137.00
16	22	0.00	22.00
17	187	0.00	187.00
18	44	0.00	44.00
19	100	0.00	100.00
20	139	0.00	139.00
21	76	0.00	76.00
22	145	0.00	145.00
23	26	0.00	26.00
24	155	0.00	155.00
> 24	0	100.37	100.37

Table A.13: `stokes.clumeq.mcgill.ca` Hourly Throughput

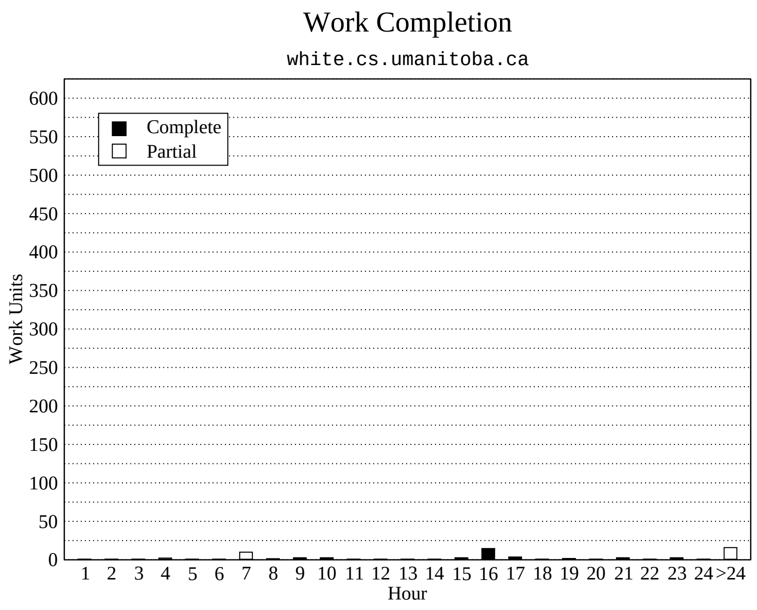


Figure A.14: `white.cs.umanitoba.ca` Hourly Throughput

Hour	**Complete**	**Partial**	**Total**
1	0	0.00	0.00
2	0	0.00	0.00
3	0	0.00	0.00
4	0	2.24	2.24
5	0	0.00	0.00
6	0	0.00	0.00
7	0	9.92	9.92
8	0	1.45	1.45
9	2	0.00	2.00
10	2	0.00	2.00
11	0	0.00	0.00
12	0	0.00	0.00
13	0	0.00	0.00
14	0	0.00	0.00
15	2	0.00	2.00
16	14	0.00	14.00
17	3	0.00	3.00
18	0	0.00	0.00
19	1	0.00	1.00
20	0	0.00	0.00
21	2	0.00	2.00
22	0	0.00	0.00
23	2	0.00	2.00
24	0	0.00	0.00
>24	0	15.85	15.85

Table A.14: `white.cs.umanitoba.ca` Hourly Throughput

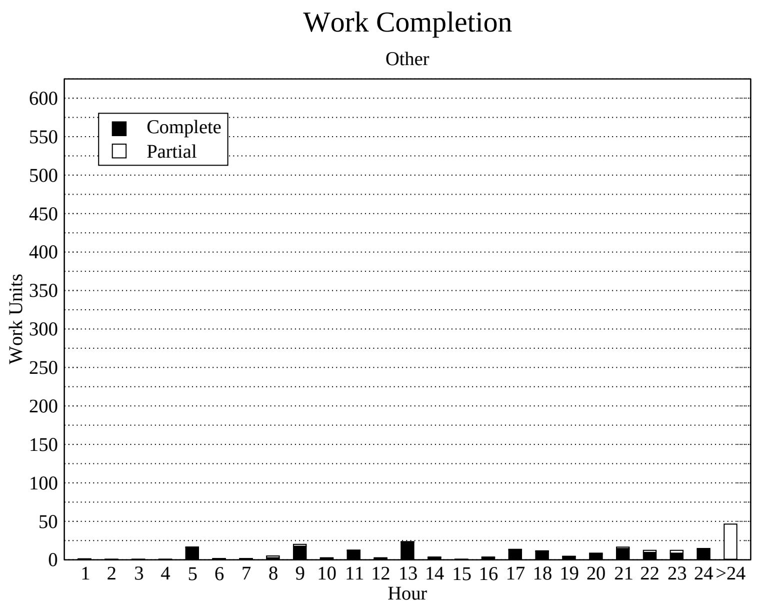


Figure A.15: Other Hourly Throughput

Hour	Complete	Partial	Total
1	0	1.31	1.31
2	0	0.00	0.00
3	0	0.00	0.00
4	0	0.00	0.00
5	16	0.00	16.00
6	1	0.00	1.00
7	1	0.00	1.00
8	2	2.99	4.99
9	17	3.11	20.11
10	2	0.00	2.00
11	12	0.00	12.00
12	2	0.00	2.00
13	23	0.00	23.00
14	3	0.00	3.00
15	0	0.00	0.00
16	3	0.00	3.00
17	13	0.00	13.00
18	11	0.00	11.00
19	4	0.00	4.00
20	7	1.65	8.65
21	14	2.49	16.49
22	9	3.35	12.35
23	8	4.29	12.29
24	13	1.73	14.73
> 24	0	46.44	46.44

Table A.15: Other Hourly Throughput