Metacomputing: An Evaluation of Emerging Systems

David Cronk[*], Graham E. Fagg[+*] and Brett D. Ellis[*]

Abstract

Metacomputing consists of the idea of connecting geographically distributed high performance computing resources in a seamless manner. This allows a single user to access a disparate set of resources from a single machine, with little or no knowledge of the underlying connections and protocols. The primary benefits of using a metacomputing system are attaining access to resources that would otherwise be unavailable, and allowing the system to hide the complexity of resource management from the user. Two current metacomputing systems that are widely recognized as leaders in the metacomputing community are Globus and Legion. This report evaluates Globus and Legion, covering a variety of criteria, including installation, maintenance, usability, functionality, and performance.

1 Introduction

The term *metacomputer* typically denotes a networked virtual computer, consisting of possibly geographically distributed resources connected by high-speed networks. Metacomputing is motivated by a need to access computing resources not often located within a single computing system. These resources are often not co-located for a variety of reasons ranging from the cost of supercomputers to the infrequency of needing certain configurations [1].

Metacomputing systems allow large-scale applications to make use of collections of high performance computing resources in a seamless manner [2]. These systems hide the complexity of managing such a system from the user, allowing the user to be more concerned with the science, and less concerned with implementation details. That is to say, the user can be working from a desktop machine while the underlying system handles the details of accessing remote resources and coordinating the computation. Such a system provides many potential benefits to the user. One very important such benefit is that the user need not learn the sometimes cryptic login and job submission protocols for each high-performance system to which he has access.

Another benefit is that with a metacomputing system a single problem may execute on multiple supercomputers simultaneously. This allows problems that are too large to execute on traditional high-performance systems to execute in a larger *metasystem*. A further benefit of this is that certain applications can take advantage of different architectures for different tasks in the problem. An example might be a simulation problem that uses a particular architecture to perform the simulation, while another architecture is used to visualize the results. This division of tasks on different architectures may even take place without the user's knowledge. This helps the user to make better use of the resources available. A well designed metacomputing system will allow the user to submit a job once, and the software will select the appropriate resources on which to execute the different tasks. This resource selection should be based on many factors, including current load of the resource and the appropriateness of the resource for the type of task.

There are currently many metacomputing systems in various stages of development, and offering various capabilities. Some of the more mature, in terms of stage of development, include Globus and Legion. Globus is a metacomputing toolkit being developed at Argonne National Laboratory and ISI USC. Legion is being developed at the University of Virginia.

[*] Innovative Computing Laboratory, University of Tennessee, Department of Computer Science, 104 Ayres Hall, Knoxville, TN-37996-1301.

[+] Project Principle Investigator. Email address: fagg@cs.utk.edu

This report evaluates emerging metacomputing technology in general, but with particular emphasis on Globus and Legion. Many criteria are used in this evaluation, including ease of installation and use, scalability, security, and others.

The rest of this report is organized as follows. The next section presents an overview of the two systems of particular interest. This is followed by an evaluation of issues relating to users. Section 4 covers usage factors with section 5 evaluating issues relating to site autonomy. Section six discusses factors relating to the future growth of metacomputing systems with the final section presenting conclusions about metacomputing technology in general and the three systems of specific interest in particular.

2 Overview

This section gives an overview of the two systems of particular interest in this report. These systems are Globus and Legion.

2.1 Globus

Globus is a joint project between Argonne National Laboratory (ANL) and the University of Southern California's Information Sciences Institute (ISI). "The Globus project is developing the fundamental technology that is needed to build *computational grids*, execution environments that enable an application to integrate geographically-distributed instruments, displays, and computational and information resources. Such computation may link tens or hundreds of these resources" [3]. A primary goal of Globus is to enable new methods of computation through a set of core services that change the manner in which resources are accessed [4]. The Globus project focuses on the development of low-level mechanisms used to implement high-level services, and techniques allowing these services to observe and guide the operation of said mechanisms [1].

Globus grew out of the I-WAY project, which was conceived in early 1995. The I-WAY project focused on developing a large-scale testbed for deploying high-performance and geographically distributed applications [5]. The I-WAY project was successful is demonstrating that large-scale geographically distributed computation was feasible, but at the same time showed that there was a lot of work needed before such systems could be used in a production code environment. The Globus project was started to address the issues of resource location, automatic configuration, scalable trust management, and high-performance distributed file systems.

The Globus toolkit comprises a set of six modules defining interfaces used by higher-level services to invoke mechanisms. These mechanisms are implemented in different environments by using appropriate low-level operations [1].

The *resource location and allocation* module provides mechanisms for expressing application resource requirements, identifying resources that meet these requirements, and for scheduling resources once they have been located. Since resource load and availability can vary greatly in a large system, applications cannot be expected to know the exact location of required resources. Resource allocation involves the scheduling of resources and performing any required initialization.

The *communication* module provides basic communication mechanisms. These mechanisms permit efficient implementation of a wide range of communication methods, including message passing, remote procedure call, distributed shared memory, stream-based, and multicast. These mechanisms are also aware of network quality of service parameters. This module is based on the Nexus communication library [6], which supports a single communication operation, asynchronous remote service request.

The *unified resource information service* module provides a mechanism for obtaining real-time information about the system's structure and status. This mechanism allows components to both post and receive information. This information includes configuration details such as memory, CPU speed, number of nodes, and network information. Instantaneous performance information and application-specific information are also included. Globus uses what they call a *Metacomputing Directory Service* (MDS) as a single, unified access mechanism for all of this information. The Globus MDS is built on the data representation and API defined by the *Lightweight Directory Access Protocol* (LDAP), defining a framework for representing information of interest to metacomputing applications.

The *authentication interface* module provides basic authentication mechanisms. These mechanisms are used to authenticate both users and resources, and are used as building blocks for other security related services such as authorization and data protection. This module is built on the *Generic Security System* (GSS) API from the IETF, which defines a standard procedure and API for obtaining credentials, mutual authentication, and message encryption and decryption. GSS is an independent system that can be layered on top of different security systems such as Kerberos and SSL.

The *process creation* module initiates computation on a resource once the resource has been located and allocated. This includes starting the executable and managing termination and process shutdown.

The *data access* module provides high-speed remote access to persistent storage. Data resources such as databases may be accessed via distributed database technology or CORBA. This module addresses the problem of achieving high performance when accessing parallel file systems and network-enabled I/O devices. Primitives are defined that provide remote access to parallel file systems. These primitives make up the *Remote I/O* (RIO) interface [7], which is based on the *Abstract I/O Device* (ADIO) interface [8].

It is important to point out that the above mentioned modules are not intended for direct use by an application. These modules are designed to be used by a middleware layer, which provides services to the application layer. For example, a middleware layer would use the resource location and allocation module to find and allocate resources for an application. This hides the details of location and allocation from both the applications programmer and the user. Such higher-level services are developed to serve as application-level services.

An example of such a middleware is the Globus implementation of MPI. The Globus implementation of MPI provides a well-known standard interface to the applications programmer. The MPI implementation makes use of the modules describe above, hiding the details from the programmer and user.

The Globus metacomputing toolkit is currently being used as part of the *Globus Ubiquitous Super-Computing Testbed* (GUSTO). GUSTO is intended to be a computer science rather than an application testbed, focused on deploying and evaluating basic mechanisms, rather than testing application performance. Globus is also part of the NASA Information Power Grid testbed, connecting a number of NASA high-performance resources across the country [9].

2.2 Legion

"Legion is an object-based, meta-system software project at the University of Virginia. From the project's beginning in late 1993, the Legion Research Group's goal has been a highly useable, efficient, and scalable system founded on solid principles… Legion is a work in progress: our team will not finish Legion but will create an "open" system that allows and actively encourages third-party development of applications, run-time library implementations, and core system components" [10].

Objects are the Legion building blocks for constructing a wide-area operating system. Legion is structured as a system of distributed objects where all the entities are represented by independent, active objects that communicate using a remote method invocation service. Object interfaces are described using an *Interface Description Language* (IDL), and are compiled and linked to implementations in a given language [11].

Legion runs on top of the operating system of each host in the system. This means that Legion does not need to manage low-level resources on each host. Instead, Legion's resource base consists of processors and storage devices. Both processor and storage resources are represented as objects in Legion. This allows for a uniform interface to hosts and storage, regardless of the underlying architecture. That is, while the underlying implementation may be different, the interface remains the same. This also allows local administrators to enforce local policy. If an administrator wishes to restrict job creation to local users, the host objects at the site can enforce this policy.

Tasks are managed in Legion by an arbitrary set of objects known as *class managers*. A class manager is an otherwise normal Legion object, which is responsible for the management of a set of other Legion objects, known as instances of the class manager. Class managers export an interface that supports standard task management operations, including creation, destruction, and queries. Class managers also act as policy makers and active monitors for their instances.

Class managers are themselves managed by higher-order class managers. This creates a hierarchy known as the Legion *domain*. There can be any number of these domains in a Legion system, making a Legion system a forest of class manager hierarchies.

Legion objects are persistent, existing beyond the lifetime of their creating program. When a Legion object is active, it can be deactivated. This causes its state to be saved to persistent storage and its containing process deallocated. If the object later must become active, its class manager automatically reactivates the object, making object deactivation and reactivation transparent to clients of the object.

Legion objects are identified by a three-level naming mechanism. The lowest level naming consists of an *Object Address* (OA) containing a list of network addresses. The middle layer naming is called a *Legion Object Identifier* (LOID), which is a unique identifier and ports, assigned to an object at the time of its creation. LOIDs are binary, variable length identifiers, which are inconvenient for user-level naming. The highest level naming mechanism is called *context space*. Context space is a hierarchical directory service, which uses Unix-like path strings assigned to objects. Legion context space does not correspond to the physical location of the object. Even though a Legion system may span multiple administrative domains, an object is identified with the same context space name regardless of its physical location.

Since files in a Legion system are simply another type of object, the file system presented to a user has no concept of location. Users are presented with the familiar concept of paths, directories, and files, with these files being globally accessible, regardless of physical location.

Legion supports a variation of remote method invocation designed to address the needs of wide-area computing, including reduced interprocess communication and latency tolerance. Legion combines a basic, low-level message-passing service, with a remote method invocation model known as macro-dataflow (MDF). This allows multiple concurrent method invocations, overlap of remote methods and communication, and the redirection of results to the objects that need them.

Security is an important concern for any metacomputing system, for both resource providers and users. The basic security provided to users is user-selectable data privacy within the message-passing layer on a per object per invocation basis. Messages can be fully encrypted, digested and signed, or sent in the clear for higher performance. Users can dynamically adjust authentication requests on a per object basis as well change the key length in the RSA public key system used.

Since all resources are represented as objects, access control and resource protection is specified at the object level. Every object has an internal method, called *MayI* that is invoked automatically when an external method request arrives. The MayI method can range from very simple to very complex, depending on the degree of security needed. An object representing a resource would have a MayI method that controls access to said resource, which is dynamically configurable.

Legion is currently deployed as part of the VANet Centurion testbed located at the University of Virginia. Centurion consists of 128 533 MHz Dec Alpha machines and 128 dual 400 MHz Pentium2 machines. Legion is also part of the NASA Information Power Grid testbed [9], as well as a resource at the NPACI SDSC site.

3 Human Factors

An important part in any decision regarding whether or not to deploy any large system is the cost in terms of human factors. This includes the amount of time that must be spent by an administrator in terms of installation and maintenance. If a system requires too much effort on behalf of the administrator for maintenance, most organizations are going to tend to go in a different direction. Similarly, if a system takes a significant amount of time for the users to learn it, or the users must put in significantly more effort with said system than with other, traditional systems, many places are going to opt to not pay the high price of deployment. Related to these issues, places considering a significant investment in a new system want to feel confident that they can receive reasonable support from the developers of the proposed system. This support includes assistance in matters of use and configuration, but also timely response to bug reports.

This section evaluates these human factors as best as could be done with the resources available during the evaluation period.

3.1 Installation and maintenance

An important human factor is the cost of installation and maintenance. This includes the time spent by a system administrator for the initial installation as well as time required to keep the system up.

3.1.1 Globus

The Globus software as downloaded from the Globus web site is not the whole system. This is something you should understand from the outset. The Globus System is intended to be worldwide. It is built on the same concept as DNS, which means your resources will be maintained on some nodes of a tree. You have a root, which is Globus. Then leaf by leaf and node by node you add organizations, sub organizations, and resources. For example, if you were part of ToBuildSomething Incorporated, you would look like the following on the Globus tree: C=US, O=Globus, O=ToBuildSomething Inc. This could be further subdivided into smaller units. If you were a university you might have another classification for department, and maybe even a research group within a department. Might be something like C=US, O=Globus, O=Unknown University, O=Computer Science, OU=Research Group. This can get complicated quickly. It helped our system administrator to think of this along the lines of DNS.

The Globus tree is referred to as the MDS tree, and is based on the *Lightweight Directory Access Protocol* (LDAP). LDAP is not Globus. LDAP is how the tree is built and maintained through directory service. SSL is also not Globus, but you have to install it as well to make Globus work. SSL is used as an authentication protocol with its optional encryption mechanism. Since Globus is something that is built with the idea of a global community, you need to understand that you have to go off-site for a number of your resources to work correctly. Your Globus install is going to involve many people, some of whom you may have little interaction with. They will, however, be the ones to hand you the methods involved in keeping the Globus system secure. While it is possible to make a system completely on your own, this is not within the design intentions of a global computer; thus it takes a lot more work. This is covered in more detail below.

<u>Installation</u>

The following is a concise description of the installation procedure for Globus. This should be thought of as an addendum to the Globus Installation Guide [27]. This reflects lessons learned from a system administrator with no previous experience with metacomputing systems. The instructions from the Globus organization are very well written as long as you plan to make a basic install. This section will cover the basic install. While the installation of a basic system pretty much follows along step by step with Globus install notes, the Globus system is anything but basic. It is important to note that being a global system it is a complicated overall picture.

Initial Pre-Installation Steps

The first thing you must do is download a copy of Globus, LDAP, SSL, and perhaps MPICH, if you plan on using MPI. Next, create a Globus user account on your system. You can call the user anything you want, but for this document, we will call the Globus user account "globus". This will be the account that owns the Globus system executables. Select where you want your globus-src and your globus-build directories. It has been observed that it is helpful if all machines involved in your Globus system have access to all of these binaries and directories during the install. Our system administrator opted to make them NFS mounted. SSL and LDAP should also be accessible on each machine involved.

Next you must send a request to the Globus organization to request your Organization's node in the Globus tree. This tree is also referred to as the *Metacomputing Directory Service*, or MDS. This is covered in detail in the install manual [27].

All of the directories above can be the same on all of your machines; each machine is required to have unique Globus-Deploy directories. This will be covered in more detail later. In our case, we choose /usr/local/globus-src, /usr/local/globus-build, and /usr/local/globus.

Compilation and Installation of Binaries

Compile SSL, LDAP, and Globus. This is fairly straightforward, and listed in the install guide. You can install SSL and LDAP as root, but you need to compile Globus as the user "globus". In fact just about everything else you do from here on out is as the user "globus". Therefore all directories involved must be writeable by that user, including the SSL and LDAP directories.

Globus-Deploy

Here's where it starts to get little more difficult. This will be at least the second time you have to deal with the outside world. You must run a script called "globus-setup" in the sbin directory of "globus -build". This will query you for information about the MDS server; its LDAP port number, your distinguished name, the directory manager (filled in automatically), and the password that you set up when you requested your bit of the node. After you enter this information, you run the "confirm" com mand and Globus will set up several files that you will need for subsequent steps. If any mistakes were made in entering the above information, this command will not run successfully. Note that you cannot successfully deploy until you run this command successfully. You cannot run the setup command until you get your information back from requesting your node. Basically, do your best to plan ahead. Assuming you have gotten through the "confirm" command, you can deploy on any machine for which you have b uilt executables. This puts executables in the globus-deploy directories that you specified in the pre-installation step.

Getting a running system

Globus authenticates users via a gatekeeper process. The gatekeeper is a Globus executable that runs as a Unix service. You will have to be root to change the files /etc/inetd.conf and /etc/services. Most Unix system administrators are very familiar with these files. This creates jobs that run as root, even though the Globus system will run as the user "globus". Globus includes an instruction list on what has to be put in these files and how to restart the necessary services. You also have to send e-mail to a CA requesting your gatekeeper's security certificate. The CA (Certificate Authority) will send you back e-mail with a certificate for that node's gatekeeper. We found that we had to e -mail a certificate request for every node on which we deployed, as we aren't using backend batch queuing software. Once you get your gatekeeper certificate, you put it in the correct directory and you have a running system. It will even restart on after a reboot since you have made changes to /etc/inetd.conf and /etc/services.

How does a user get access to all of this?

For any user to be able to use Globus, he must first obtain a Globus certificate. The steps taken to do this are covered in Section 3.2.1 below. The system administrator must maintain a grid-mapfile on each machine in the Globus system. This file contains mappings between Globus user certificates and local user names. Only users in a machine's grid -mapfile can access the machine via Globus. Once you have completed the above steps, you have a working Globus system that users who are in the grid-mapfiles can access.

Installation Problems and warnings

The Globus install tends to proceed cleanly and few problems were encountered. However, there is a lot of outside intervention required. For this reason, deploying Globus on a large number of machines is time consuming. Also if you attempt anything but a basic install, expect headaches. What follows is a discussion of the problems encountered and lessons learned by a novice Globus administrator.

NFS

In our case compiling NFS proved to be problematic. The compile of the Globus system is a longone, and it is fairly intensive. Our experience was with NFS on a 10Base-2 backbone. Once we compiled on the local machine all of those errors disappeared. This problem will probably not present itself for most institutions, but for those with slower networks that might be something they could look at.

Outside intervention

Since the Globus development team manages the security mechanism for you, you must obtain security certificate through the Globus organization, or some other third party certificate authority. These certificates are encryption keys used for mutual authentication between users and resources. These certificates are recognized system wide, allowing a user to obtain a single certificate, which can be used to access Globus systems worldwide. This means that with your one certificate and key, you can use any Globus system on which you have been placed in the grid mapfile. However, for this to work, all resources that are part of the Globus system must have a gatekeeper with a certificate of its own. This means that **EVERY** machine on which you want to run a gatekeeper requires a separate certificate. This requires tremendous effort for a large system consisting of individual machines as in a cluster. For example if you have a 128 node cluster you have to send 128 e-mails for certificates, wait for 128 certificates to return, and then successfully put the 128 certificates on the 128 machines. While this provides for better security, it does require a lot of effort on the part of the administrator. If alternately you need to support an MPP with some type of single job submission point, you only need a single certificate for this system no matter how many nodes it consists of.

Passwords

Your site's administrative password is stored in a p lain text file in the Globus-Build directory. We discovered this by chance. While it is only readable by root, the fact that it is plain-text and non-encrypted is a source of concern.

Non-basic installation

Initially, we attempted to install a freestanding Globus system. That is, a system isolated from the rest of the Globus community. We wanted to maintain our own local MDS and act as our own Certificate Authority. This was found to be very difficult; in fact, we were unable to accomplish this. This is outside the planning aspect of the Globus project, as it is not in the spirit of the project. It is important to understand that your local system must access two outside entities to function correctly; the MDS and the

CA. The MDS is the directory service that organizes the many sites and units that form a tree. There are beta instructions on the Globus web-site to set up your own MDS tree. This led us to believe that it would be possible to set up an isolated system. That was definitely a wrong assumption. We successfully got a MDS system up and running, although it was no easy task. This involves downloading Netscape's Directory server, installing it on a machine in your network, and gaining at least a rudimentary understanding of the LDAP protocol. You must enter the correct LDAP control strings and it will control your system. LDAP string are: Distinguished name C=US, O=Globus, O=Your University, OU=Your Department, etc. It is extremely important that if you do decide to undertake building your own MDS tree, you have C=US, O=Globus in that order before you enter the rest of the string. The Globus software is built only to understand distinguished names in that order. However, even with your own MDS, you still don't have a Certificate Autho rity. This means you cannot create your own certificates. In order to have a truly isolated Globus system, you must be able to act as a CA. The Globus group does not encourage this and we received very little assistance in attempting to become a CA. In fact, we were discouraged from doing so. It has been reported that there are some sites acting as Certificate Authorities, so it is believed to be possible. However, unless you are prepared to spend a lot of time setting up the MDS and CA for your site, you will need to be part of the entire Globus community. This means relying on an outside organization for issuing security certificates and allowing your resources to be listed in a publicly viewable MDS tree.

Notes

After the initial draft of this report became available, the Globus Project members at ISI became more willing to discuss and support independent MDS and CA usage. We are currently actively testing methods of building such independent authorities with the Globus team. We also are aware now that the SCSD NPACI site is acting as an independent CA for their Globus users. This type of support is important as some sites may simply be unwilling to use a system that requires their resources to be publicly known and controlled.

3.1.2 Legion

Compilation of the Legion system under Linux has been found to be straightforward if your OS installation is standard. We have not testing compiling under Windows or other versions of Unix as yet. The only problems we encountered with compiling was the need to have the Korn Shell (ksh) available (in /bin), and having the Unix Tar utility not unpack all the source files correctly. This later problem can be solved by using the GNU version of Tar, which handles very long path names correctly. Earlier attempts at compiling failed due to non-standard OS features added to our systems and we were forced to download pre-compiled binaries. These pre-compiled binaries could cause linking problems when compiling applications as is discussed in Section 3.3.2.

Installation

The following is a concise description of the installation procedure for Legion. This should be thought of as an addendum to the Legion Installation Guide [12]. This reflects lessons learned from a system administrator with no previous experience with the Legion metacomputing system.

Initial Pre-Installation Steps

Create a legion user Unix account. The home area for legion should be viewable by all machines involved in the Legion system. It is highly recommended that you run all system Legion processes as this user. In this example we will assume that the user login created was "legion". There should be either be a ssh mechanism or an .rhosts file in legion's home area that contains machines 1 -4 so that legion can rsh from machine to machine with no password. Theoretically, all of the binaries would reside in your home area and you can make OPR directories on machines in /tmp or scratch areas. However, it seems to make more sense for system administrators and be less confusing to make a separate "legion" user.

Obtain a copy of Legion. You can get source files, as well as the binaries from the Legion web site after submitting a license agreement. This guide assumes you downloaded legion binaries. Unpack the binaries into an exported partition on machine0.

Choose LEGION and LEGION_OPR environment variables. The LEGION environment variable is where the binaries from the previous step are located. This should be a NFS'd (or other variant) file system. It is convenient for users to use /usr/local/Legion. This directory's contents are architecture dependent. Note: multiple architectures can share this directory, they will each have folders off of $LEGION/bin. Note: there is no apparent reason not to locate the legion binaries in the legion home area. The LEGION_OPR directory is the working machine directory. I.e. this is where all of the system files, machine state, etc is located once you start a machine. It should NOT be the same directory as LEGION. In our case we choose /usr/local/OPR. Each machine maintains its own DISTINCT OPR directory that should not be NFS mounted if you want good performance. The legion account set-up above should own both directories, /usr/local/Legion and /usr/local/OPR. Keep in mind the only information the general user will need is the setup* files that will be located in the OPR directory on the BootstrapHost. This will be covered in more detail below.

Installation

Choose a "bootstrap" host. This host is the ONLY machine you will run the following installation steps on. The bootstrap host is responsible for getting the appropriate OPR files to other machines. You will get various questions on what parts you want to start up. For our installation we answered a Y for yes to all.

1. setenv LEGION /usr/local/Legion
2. setenv LEGION_OPR /usr/local/OPR
3. source $LEGION/bin/legion_env.csh
4. legion_setup_state
5. legion_startup
6. legion_initialize
7. source $LEGION_OPR/legion_context_env.csh
8. legion_init_security (you will have to give a password for legion admin)
9. legion_login /users/admin (give password as specified in step 8)

You have a single working Legion machine on your bootstrap host with one user. A little more about Legion: The user readable form of the Legion context space is viewed as a tree hierarchy. At this point you can do a legion_ls and get the root of the legion machine. As above, users are located in /users, so when they log on they log on as /user/USERNAME. There will be a hosts directory etc. You probably want to make a few users at this point. This is done as follows:

10. legion_create_user

This will create a user instance in the /home area of the Legion machine. You will also have to give each user an initial password. You can add users at any point to a Legion system.

11. legion_logout

This completes the steps necessary to create a Legion system. This system, however, only consists of a single machine. The following details the steps necessary to add machines to the Legion system. The first thing you need to be able to do is login as the Legion admin user.

Logging on…

You first should log onto the machine on which Legion was originally installed. Login as you normally would, as yourself. Then copy ~Legion/setup.{sh/csh} to your home area. This copies various files to your home area. For csh users, simply "source setup.csh". That's it, all of your environment variables will be set, and you'll have an OPR directory of your own. This is where all of your personal files will be located. Now do a legion_login /users/USERNAME and you are on the legion system, where USERNAME is created in step 10 above. You can run legion programs from this point on. When you are done, run the command legion_logout.

Adding Machines

You must then login to the bootstrap host (the machine where the initial Legion installation was performed) or any other once Legion has more than a single host, as the Unix *legion* user. Source the setup.csh file you copied from above. To add machines do the following:

1. legion_login /users/admin
2. legion_startvault machineX.yourdomain.com
3. legion_starthost machineX.yourdomain.com /vaults/vault-machineX.yourdomain.com
4. Repeat steps 2 and 3 for all machines you want to run Legion on besides your BootstrapHost. In our case machineX was machine1, machine2, machine3, and machine4.

These commands assume that you have some sort of rsh access to all machines and write permissions on the /usr/local/OPR directory on all of those machines.

This completes the steps necessary for adding machines to your Legion system. You have a working Legion system with multiple machines.

Installation problems

The following represents some of the problems (and solutions) encountered by a system administrator with no previous experience installing metacomputing systems. No claims are made that other administrators will encounter these problems.

More than just BootstrapHost at Install

Another source of confusion in earlier documentation was the description of how to add machines to a Legion system. The documentation stated that any machine to be added should have a "Legion system already installed". This can be taken as meaning that each machine should have a complete installation as described above. This is incorrect as it leads to a broken installation. The main problem is that there is no indication from the system that this is incorrect. Users will have problems working with the system and there will be no information given that will lead to a correct diagnosis of the problem. Correct translation of each machine must have a "Legion system already installed" is each additional machine must have working Legion binaries available. The Legion 1.6.4 documentation corrected this statement, and the Legion team has promised that an install validation suite will be developed to assist in testing if a given installation is correct.

User's environment

Another source of confusion was the belief that the environment is set up the same for general use as it is for installation. From above, the following steps should be taken to set up the environment.

1. setenv LEGION /usr/local/Legion
2. setenv LEGION_OPR /usr/local/OPR
3. source $LEGION/bin/legion_env.csh
4. source $LEGION_OPR/legion_context…

These steps should **not** be performed by the general Legion user; only the Legion administrator should set the environment this way. If all users perform step two, every user will share the same OPR directory. This is not the correct setup. All setup files needed by the user are placed in LEGION_OPR on the BootstrapHost. See the login description above for an example. This sets up a separate OPR directory for each user, rather than every user attempting to share a single OPR directory.

Maintenance problems

If your Legion bootstrap host should be shutdown in an unsafe manner (such as power loss), you must reinstall the **full** installation. But please note that user application compilation is performed under the Unix (or NT) file system and so all a user has to do is re-register his/her applications within Legions context space. As user data itself can be linked into Legion space from Unix, loss of any data is usually not a problem. This must be done from the bootstraphost: the machine on which Legion was originally installed. Individual host failures are automatically recovered from by the system as part of the Legion Host Object Recovery algorithm.

Bootstraphost failure is not automatically recovered from, and thus this host should be a more reliable and controllable host. A good method for avoiding a complete reinstall under these conditions was taking a snapshot (zipped tar file) of the bootstraphost's OPR directory once it had been started, and then just copying these files back after any kind of failure.

3.2 Ease of use

Another important human factor is the *ease of use* issue. If a software system is difficult for users to learn and make use of, the chances are an organization is going to resist using it. Ease of use includes learning the system in the first place as well as getting the system to perform once it has been learned. This includes completeness and accuracy of user documentation. Ease of use also involves understandable error messages and being able to debug programs. This section focuses on issues pertaining to using these systems to run MPI programs.

3.2.1 Globus

In order to use Globus, a user must first obtain a Globus certificate. A Globus certificate acts as a "passport". You go to the proper authorities to establish your identity, and they issue you a certificate and private key. This means you do not need to reestablish your identity at each location accessed. As a consequence, you need only enter your password once per Globus session [13].

To obtain a Globus certificate you must first log onto a machine with Globus installed. Once logged on set the environment variable GLOBUS_INSTALL_PATH to the Globus build directory. Your Globus administrator can tell you where this is. An example is /usr/local/globus-build. Once this variable is set, execute the grid-cert-request tool. Your administrator can tell you where this is. An example is:

```
$GLOBUS_INSTALL_PATH/tools/x86-pc-linux-gnu/bin/grid-cert-request
```

Running this command will print instructions for getting your Globus certificate. The next step is to mail the request to the Globus organization. In a Unix environment this can be done by entering

```
%cat ~/.globus/usercert_request.per | mail ca@globus.org
```

You will then receive your Globus certificate via email. Save this email to the file ".globus" in your home directory and forward this certificate to your Globus administrator and you will be added to your local Globus system.

Finally, change the permissions in your .globus directory.

```
%chmod 400 ~/.globus/userkey.pem
%chmod 444 ~/.globus/usercert.pem
```

You are now ready to use Globus.

Using Globus

After obtaining a Globus certificate, you can then log onto a machine that is part of your local Globus system (the system on which you obtained your certificate). Next make sure that your environment is set up properly. This can be done by first setting the environment variable GLOBUS_INSTALL_PATH as discussed above. Once this variable is set, you can set up your environment using setup files provided. For a csh shell you can enter the following:

```
%source $GLOBUS_INSTALL_PATH/etc/globus-user-setup.csh
```

Once this is done you are ready to use Globus. Experience dictates that the first command you should run on any new Globus session is "globus-setup-test". This will ask for your password and will test the system. If any part of the system is not working, this command will let you know. Failure to run this command may lead to great aggravation when you have trouble with other Globus commands and cannot determine why. Once you have confirmed that the system is up and running you need to initialize a proxy. Running the command "grid-proxy-init" does this. This generates a proxy, which is valid for 12 hours by default. While the proxy is valid you will not need to enter your password on any of the grid machines. You can check the status of your proxy by entering "grid-proxy-info –all". This command will tell you if you have a proxy and how much time is left before it expires. You can also destroy your proxy before it expires by entering "grid-proxy-destroy".

From this point, there are several ways to execute Globus jobs. Jobs can be run by explicitly identifying the resource on which to run the job, letting the Globus scheduler decide as under MPIRUN or submitting jobs specified using the Resource Specification Language (RSL).

Running MPI jobs using Globus

Globus uses its own MPICH for MPI jobs, referred to as MPICH-G, although it does also support the use of a systems native vendor MPI implementation. This is the MPICH implementation of MPI using the Globus device. In order to run MPI jobs using Globus you need simply to recompile the application using the MPICH-G compilers, include files, and libraries. This requires using mpicc and mpif77 as your C and Fortran compilers respectively. These can be found in the bin directory of the MPICH-G installation. The scripts mpicc and mpif77 automatically use the proper include files and MPI libraries. Once your application has been recompiled, you are ready to run the job using Globus. Make sure your path is set to use the mpirun command located in the bin directory of the MPICH-G installation.

The first thing you need to be able to run your job is a file that identifies resources. The simplest way to do this is have a file named "machine" in the same directory as your executable. This file has the form:

```
manager/resource-name [number of processes]
```

Number of processes is only needed when running on multiple hosts. An example file may look like this:

```
torc1/jobmanager-fork 1
torc2/jobmanager-fork 1
```

At this point you can use mpirun to execute your program. While there are many details hidden from the user, conceptually this works the same was as any other MPI programs you run. It is also easy to run jobs on machines that do not share a file system or machines that use a batch queuing system. See the Globus User's Guide for mo re information [13].

The Globus user documentation seems to be both straightforward and accurate. No problems were encountered using the documentation as a guide for compiling and running MPI programs under Globus. Most of the rest of the Globus documentation appears to be as helpful as that pertaining to MPI programs. The most difficult aspect of the Globus documentation appears to be dealing with the Resource Specification Language (RSL). However, it appears that most users can avoid using the RSL directly due to most of the Globus commands hiding the details of the RSL from the user. The RSL was not directly used in the work performed for this evaluation since the Globus *mpirun* command generates a RSL script used by the scheduler. At a later stage of testing, however, the GRAM was found to add a host that had once appeared in the Globus configuration, but had been removed by this point in the testing. The only way to remove this host from the scheduler was to edit the RSL and change this phantom host to a real host. Also, a standalone user's manual and system administrator's manual are not yet available.

Perhaps the most difficult aspect of working with MPICH-G under Globus is in terms of debugging. It seems that short of inserting flush statements, Globus does not print to standard out until after the program has terminated. Moreover, it seems this output does not show up unless the program terminates gracefully. This makes it difficult to isolate where a job dies if it terminates abnormally. Additionally, if a job hangs and must be terminated from the command line, there is no output to aid in determining in what part of the code the job hangs. This can make debugging a large application much more problematic.

3.2.2 Legion

To be able to use Legion the administrator must first create a Legion (not Unix) user account for each new user by calling the legion_create_user scripts, which are very much like the more familiar add user scripts under Unix.

Using Legion

To use the Legion system you must first login to the Legion system using the legion_login script. The path to this script is normally stored in the OPR directory of the bootstraphost under setup.sh/csh etc.
Once you have logged on, any sub-shells you call from the current shell are also logged on as the logon information is stored in the shell's environment. This is different from Globus, which stores information in the user's ".globus" directory.

The user must also be cautious about either locating files in the correct filesystem so that running executables within Legion can see these files or importing them at run-time, as normal application I/O calls use Unix services. Executables running under Legion change their working directory (CWD) to a temporary cached OPR directory, and so the application had to be changed to use either absolute paths or explicitly change its paths to correct for this unless files where imported. Two methods are available for importing files. In the first the user could export his/her Unix filesystem for a temporary period to Legion context space by using the legion_export_dir script or more permanently by using legion_import_dir. In the second method, the legion_run command arguments can specify files to move to and from the CWD during the execution. This second method is the simplest and most recommended.

Each executable run within Legion must be of the form of a Legion object with its own unique LOID. Legion provides multiple methods to register sequential (legacy) and parallel executables as well as Mentat programs.

The last difficulty for a user switching between context space and the host operating environment is the handling of standard or terminal I/O. Output from a running Legion application is not automatically sent to the users host operating systems terminal, but rather to a terminal (TTY) within the Legion context space (/home/user/tty). For users to catch any output from this object, they have to monitor it using the legion_tty script. The administration user can potentially monitor any user's Leg ion terminal if they have not explicitly changed permission on the TTY object.

Running MPI jobs under Legion

Creating and running an application under the Legion version of MPI takes four stages:

1. (1) Compile and link with the Legion MPI libraries
2. (2) Register the executable with Legion
3. (3) Import/Export files into context space if needed, as well as monitor the user's Legion TTY if the user wants any console output!
4. (4) Start the job (specifying any files to import/export during execution if needed).

Compiling an application for MPI under Legion is straightforward, unless your using incompatible libraries as can happen with the Legion Precompiled binaries. Linking with the Legion libraries is performed by the legion_link script, which acts in much the same manner as MPICH's mpicc script by automatically setting up all required library paths etc.

Before a MPI job can be run, a context object version of it must exist in a Legion vault/context space. The legion_mpi_register script performs this function, and takes three arguments. The first argument is the class name, or the user-friendly name within context space that is used to reference the LOID of the persistent state of this object, from which running instances can be created. The second argument is the full path to the executables so that it can be copied into context space. The last argument is the architecture type, such as linux or NT for example. This is used to select the correct object for each host on which the job runs.

Once registered a MPI object can be invoked by calling the legion_mpi_run script that has an argument list very similar to that of most generic MPIRUN utilities. Commonly used arguments include number of processors, host file of target hosts, debug and fault tolerance levels and any files to import/export. One argument that was broken was the set working directory option, which we initially worked around until the Legion team fixed it during this evaluation.

Once a user understands how the Legion context space works, using Legion to run MPI jobs is no more difficult or verbose than under Globus or even a PBS/LSF scheduler at a central computing facility. The problems associated with multiple filesystems are much the same as users already encounter when having to write batch job control scripts that move data files into and out of scratch and other temporary storage systems on currently used MPPs. Most of these aspects can be ignored by using the IN/OUT arguments to the Legion run scripts.

Additional Features of Legion

Under Legion 1.6.4 a new form of host object was introduced known as the Process Control Daemon (PCD) [38]. This daemon would run as root (started by inet) and would allow Legion jobs to be run under different Unix user ids much like how jobs under the GRAM in Globus run under the users Unix id. This was introduced to allow for better tracking and logging of user usage of resources at computer facilities such as those at SDSC NPACI. For the user this would be transparent and they would submit their MPI job as before.

3.3 Assistance and support

A final human factor that may have tremendous impact on an organization's ultimate decision whether or not to deploy a large software system is assistance and support. Assistance and support pertains to response to questions and attention to bug reports. An organization wants to feel confident that their questions will be answered promptly and that bug reports they generate will receive reasonable attention. This is also often times the most difficult criteria to evaluate. This is because it is often impossible to know how much time people at a remote location are putting towards your concerns. It is often impossible to judge the difficulty of an issue. An issue that seems fairly simple may in fact turn out to be very difficult. In situations like this it may seem you are not receiving the attention you feel you deserve. Similarly, an issue that seems extremely complex may be an issue the developers have dealt with before. In this situation, what seems like a quick answer may in fact have sat around for a while before someone decided to let you know what the problem was. With this in mind, the rest of this section only reflects experiences from the work performed for this evaluation. There is no way to say that others would have similar experiences.

3.3.1 Globus

One of the first things we wanted to do in terms of Globus was to set up a local Globus system in isolation from the rest of the Globus community. It was believed that many organizations considering the use of a metacomputing system would prefer, if not demand, that they be able to maintain it locally. This was not possible with the typical Globus installation at the time of this evaluation. This is because, to be able to use Globus, all your resources must be registered with the central MDS tree located at Argonne. This is true even if you are not supplying resources outside of your own organization or using resources from outside your organization. It was felt that this would be unacceptable to many organizations.

When the Globus development team was first contacted about the possibility of setting up a locally maintained MDS, they were very attentive and very helpful. However, there was a bit of miscommunication. They thought we wanted a local MDS that pointed to the MDS tree located at Argonne. This would mean our resources would still need to be registered with the Argonne MDS, and publicly visible. Additionally, Certificates would still need to be generated at Argonne. Once we explained that we wished to have a Globus system isolated from the rest of the Globus community, the cooperation lessened considerably. The Globus developers insisted that that was not the way Globus was supposed to be used and preferred not to help us to get our system set up in such a way. They were still very helpful in many other ways, and were very helpful in getting our system up and running, but we were never able to get the system running in isolation. As indicated earlier, we are now co-operating in developing methods to do this independent installation, which includes producing documentation for general release.

Once a typical Globus system was up and running, we tried to get a large MPI program running under Globus. At first it did not run, though it worked fine under MPICH-ch_p4. This strongly suggested it was a bug in the communication system, namely MPICH-G, the Globus implementation for MPICH. Since there was strong evidence that the bug was with Globus, email was sent to the Globus development team. After a week with no response, a follow-up email was sent regarding the problem. This finally resulted in a response where the Globus people were willing to assist us. However, the MPICH-G bug was eventually discovered on our own and we supplied a work around. Two additional Globus bugs were discovered in the process of our testing.

Bug reports were sent in for all bugs discovered several weeks prior to this writing. At the time of this writing, no response has been received.

3.3.2 Legion

When downloading Legion you have the option of downloading the source code and compiling the system or downloading pre-compiled binaries. It had been suggested that it is difficult to get the Legion source code to compile, although we found this to be to the contrary since. For this reason we initially downloaded and installed Legion using the pre-compiled binaries.

Shortly after installing Legion, problems were encountered getting simple programs to compile. After considerable effort, email was sent to the Legion developers requesting assistance. Response time was excellent and it was explained that the primary problem seemed to be an incompatibility between the compiler used to build the Legion binaries and the compiler being used to link to the Legion libraries. This initiated a series of emails where we attempted to discover what compiler had been used to build the Legion libraries. It was decided by both parties that we were not going to be able to get programs to compile with the installation in use. It was requested that the Legion developers re-build the binaries using a compiler to which we had access. The Legion team was more than happy to oblige and within twelve hours of the request we were supplied with a newly built Legion system.

Subsequent efforts to get assistance from the Legion development team met with mixed results. Simple Legion commands were failing and email requesting assistance went unanswered. However, our situation was explained, we received enthusiastic support from the Legion team. This even extended to a temporary account to our site being granted to the Legion developers. They vigorously investigated our Legion setup and discovered several major mistakes in our installation. After several days of working with the Legion developers we were able to get Legion properly installed on our system. It is believed that without their assistance we would still not have a properly installed system.

3.4 Summary

The cost of installation and maintenance tends to increase with the size and complexity of the software system. Globus and Legion are both considerably large and complex systems. This size and complexity are reflected in their relative costs of installation and maintenance. While Globus showed to be very difficult (undoable even) to install in an isolated environment, once it was accepted that a typical installation was necessary, the installation was fairly straightforward. The only maintenance concerns that arose involved adding new users to the system and maintaining grid mapfiles. The Legion software, on the other hand, was trivially simple to install since it was downloaded as tarred binaries. This ease of software installation was offset by the system installation and maintenance costs. The documentation for system installation was somewhat vague, although this has been rectified in Legion 1.6.4. However, by following the instructions in this report few problems should be encountered. These system installation instructions were written with the knowledge gained from our various mistakes. Maintenance costs include the need to maintain a compiler compatible with that used to build the pre-compiled binaries and the need the reinstall of the OPR directory during a non-graceful shutdown of the bootstrap host.

Ease of use really depends on what you want to do. This report focuses on running MPI jobs in a metacomputing environment, so ease of use pertains mostly to this end. Globus is fairly simple to use. Users need only to obtain a Globus certificate a single time, and then are ready to go. In terms of MPI programs, a basic drop- in approach is sufficient. No alteration to the actual code is necessary. Legion is only slightly more difficult to use. Other than the need to use "legion" commands, such as legion_ls, it is not that difficult to run simple MPI programs. Porting ordinary programs to use Legion features or the Mentat directives for parallelism takes much more effort due to the object-oriented design of the system. If a simple execution of a job is required, the executable only needs to be wrapped up and placed in context space with the legion_register_program / legion_mpi_register calls, with no code modifications necessary.

If the user needs only to access remote data via Unix IO calls, Legion applications have the advantage in that files can be imported and exported at run-time using arguments to the Legion run scripts. Globus applications would need to be modified to make explicit GASS requests instead.

4 Site Autonomy

4.1 Security

Security is perhaps the most important aspect of a metacomputing system, for both resource providers and users. Resource providers must be confident that their systems will not be compromised by rogue users. While this may include having control over who has access to their computational resources, it most certainly includes protecting the integrity of their systems. Resource providers will most likely reject a system that increases the possibility of a site coming under attack.

Users are also concerned with the issue of security. Some users may insist on guarantees that data can be transferred in a secure manner. Users must also feel that their data is secure from other users, both on their local system and on remote systems. Just as resources must be safe from rogue users, data must be protected from rogue hosts.

4.1.1 Globus

The primary security goal of the Globus project is to provide security at least as good as that at participating sites. Additionally, the Globus developers feel that they must "provide a precise definition of what it means for the system in question to be secure" [22].

The Globus Security Infrastructure (GSI) focuses just on authentication and is steered by two problems not commonly addressed by standard authentication technologies. These are local heterogeneity and a need to support N-way security contexts. That is, the GSI cannot change domain specific security protocols and must enable the establishment of a security relationship between any two processes in a computation [22].

Local heterogeneity is handled by mapping a user's Globus identity into local user identities at each site. This requires users to have accounts on all resources they utilize, which complicates system administration. These local user identities can then be used for local security protocols, such as Kerberos. N-way security contexts are supported by users needing to authenticate once per computation. This authentication generates a credential that allows processes created on behalf of the user to access resources with no additional user intervention [22].

All GSI security algorithms are coded in terms of the *Generic Security Services* (GSS) standard. GSS defines a standard procedure and API for obtaining credentials, mutual authentication, and message-oriented signature encryption and decryption. GSS is independent of particular security mechanisms and can be layered on top of different security methods, such as Kerberos and SSL [22, 23].

The current GSI implementation supports both a plain-text password system (similar to Unix rlogin type authentication) and public key cryptography based on SSL. GSS supports a negotiation mechanism allowing GSI to support both mechanisms simultaneously in the Globus environment [22].

The GSI is based on a certificate system. Users must have a X509 certificate that is assigned by a Certificate Authority (CA). This certificate contains the RSA public key and the signature of the certified CA. The user's private key is saved in a separate file and encrypted using the user's pass -phrase (password). The user must also have a copy of the trusted certificate of the CA. Each resource also has a copy of the trusted certificate of the CA as well as a certificate of its own. These two certificates are used for mutual authentication. When a user submits a job request to a resource, the user and resource exchange certificates and perform the SSL protocol. Each checks the CA signature contained in the other's certificate against the signature on the locally stored CA certificate. The resource also checks the user's globusid against a list of permitted clients. This means the client must be in the Globus mapfile on the system being used. If everything checks out, the job request proceeds [23].

The GSI also supports the use of proxies. If a user creates a proxy, this proxy can be used to run multiple Globus jobs without needing to re-enter a pass-phrase. Proxies essentially allow job submission without pass-phrase protection, so proxies must be kept secure at all costs. This aids in ease of use, but can potentially compromise the security of the system [23].

As mentioned in Section 3.1.1, your site's administrative password is stored in a plain text file in the Globus-Build directory. While it is only readable by root, the fact that it is plain-text and non-encrypted is a source of concern.

Authorization is performed locally. A Globus user must have an account on any system accessed. This allows local authorization policies to be strictly adhered to even in a Globus environment. Once a user's identity has been authenticated, the globusid is mapped to a local user id, which is used for local authorization.

4.1.2 Legion

Legion is designed to run across multiple administrative domains. This implies many things from a security standpoint.

- Legion cannot control security at a participating site.
- There may be malicious users in the system.
- There may be sites "pretending" to be Legion sites.
- Different users and sites have different security requirements.

The Legion security model offers no guarantee of security. The model stresses the following points:

- Be as precise as possible about the degree of confidence a user can have.
- Make that confidence "good enough" and "cheap enough" for an interestingly large selection of users.
- Provide a context that allows the user to gain the additional confidence they require, with a cost that is proportional to the added confidence they get.
- Some users will not gain the necessary confidence and will simply opt not to use Legion.

There are three basic principles to the Legion security model. These are: do no harm, buyer beware, and small is beautiful. "Do no harm" has two impacts. First: minimize the possibility that Legion will provide an avenue via which an intruder can do mischief to a remote system. Second: minimize the possibility that a user's data may be compromised by a rogue Legion host. "Buyer beware" suggests that the remote system is responsible for ensuring that they are running a valid copy of Legion, and users are responsible for their own security. This can be inexpensive in the default case, but the user has the ultimate responsibility to decide what policy to enforce and how vigorously to enforce it. "Small is beautiful" comes from the fact that one cannot absolutely, unconditionally depend on Legion to enforce security. Thus, there is no reason to invest it with elaborate mechanisms. The simpler the model, the less likely a corrupted version can do harm [20, 24, 25].

Legion is an object-based system where all resources, including processes, files, and hosts, are represented as objects. For this reason, the unit of protection in Legion is the object. Every Legion object is responsible for its own security and protection. The primary feature that supports this is that every class must define the member function MayI. Legion automatically calls MayI before every member function invocation. A user may allow this MayI function to default to "always ok", inherit a MayI from a trusted class, or write a new mechanism if the situation warrants it. This allows the user to decide on the amount of protection desired. Additionally, an object can include code in every method to determine the identity of the caller and whether or not the caller has a right to make the call [20, 24, 25].

For performance reasons the MayI function need not be called on every member function invocation. The default case of "always ok" can be optimized for zero overhead. Additionally, MayI returns a *license*, which can be used to determine when MayI needs to be called again. These licenses are cached in the stored object's address space, protecting against augmentation by the caller [20, 24, 25].

Security policies can also be enforced externally, although this is not done often in practice. The user can define an object called a Security Agent (SA). If the security agent field of the environment is not NIL, then the user defined SA is used for external security enforcement. This means A invoking a method in B is converted to A invoking the SA member function "pass". This member function of the SA decides if the method invocation is allowed. The member function "pass" is written by the user and can enforce very light to very rigid security measures [20, 24, 25].

This is all good provided objects can identify other objects (remembering that users are objects in Legion). This is done by the use of Legion Object Identifiers (LOIDs). LOIDs include a security field that is actually a X509 certificate. By default, each user has a signed X509 certificate. These LOIDs are used to authenticate objects and to control access to resources. Legion also provides what are called "credentials".

Credentials are a list of rights granted by the credentials maker. A credential specifies the period of time it is valid, who is allowed to use it, and the rights associated with it (which methods may be called on which methods or classes of methods). Basically, a credential passes rights to objects that don't naturally posses these rights. An example might be, object A uses a credential to grant object B the right to call object C's method M. As long as object A has a right to call objects C's method M, object B now has the right to call object C's method M, for as long as the credential is valid [24]. The current Legion X509 security forma t is experimental and is not currently included as part of the public release.

Legion also provides for communication security. Legion provides the following security modes:

- None: encrypted credentials, but no message encryption or digest.
- Protected: Message digest to ensure message integrity and encrypted credentials
- Private: Encrypt the entire message.

4.2 Resource management

Resource management in a metacomputing system has two aspects. The first is, how does the metacomputing system select and manage the available resources? The second is, how much control does a local system administrator have over his or her own resources?

For a metacomputing system to be successful, it must have the ability to manage resources in an efficient manner. Additionally, there must be mechanisms for selecting resources to use in an intelligent manner. These resources should be selected in a manner that attempts to minimize the total execution time of the application.

Local system administrators are going to be unwilling to deploy a metacomputing system if it means losing control of their resources. There must be simple mechanisms for sites to maintain autonomy. This autonomy must be in regards to controlling who has access to the system as well as the ability to enforce all local policies in terms of resource use.

4.2.1 Globus

Globus uses an extensible *Resource Specification Language* (RSL) to communicate requests for resources between components. This architecture uses resource brokers, resource co-allocators, and resource managers. Resource brokers take high-level RSL specifications and refine them into more concrete specifications. These refined specifications completely specify the location of the required resources. The resource broker then passes these refined specifications to resource co-allocators.

Resource co-allocators coordinate and manage the resources at multiple sites. A resource co-allocator breaks a multi-request into its constituent elements and passes each component to the appropriate resource manager. The resource manager is responsible for translating the request into a form recognizable by the local, site-specific resource management system.

The resource manager in Globus is called a Globus Resource Allocation Manager, or GRAM. The GRAM is responsible for processing RSL requests, and either denying the request or creating the appropriate processes on the local resource. Process creation is often performed by GRAM interfacing with a local scheduler or resource allocator, such as Condor, LoadLeveler, or others. The GRAM also enables remote monitoring and management of jobs and periodically updates the MDS information service with information about the status and capabilities of the resources that it manages.

More information about the Globus resource management architecture and implementation can be found in the documentation [2, 22].

Site autonomy is preserved in a Globus system by a number of mechanisms. The interface between GRAM and local schedulers, discussed above, provides a way for local administrators to maintain control over their resources. Policies that are enforced by these systems are automatically enforced within Globus. An additional feature that supports site autonomy, but also restricts flexibility is the need for users to have accounts on all resources that they access. This gives a local administrator control over the resources used by Globus users in the same way they have control in regards to their local users. Finally, in order for a user to access a given resource, that user must be in the mapfile maintained for that resource. This allows administrators to limit the resources available to individual users. In the case of large numbers of users, this can lead to considerable effort to maintain.

4.2.2 Legion

Resource management in Legion is completely decentralized. A Legion system is made up of domains, which are often defined in terms of administrative boundaries. The resources maintained at a particular site often make up a single domain Domains are managed by collections, schedulers, enactors and individual resource objects such as host and vault objects [39]. By controlling the interaction of these objects site autonomy is maintained. An organization may decide to implement its own objects, giving the organization complete control of all its resources. Alternatively, an organization may trust another organization's object classes, and choose an instance of that class to manage its resources. Organizations can even choose to put their resources under the control of another set of objects. The use of objects and collections gives participating organizations as much control as they want [14].

Legion uses a basic philosophy that scheduling is a negotiation between autonomous agents, a consumer and a provider. Legion provides simple, generic schedulers, but also allows applications to provide application specific user level schedulers. The components of the Legion scheduling model are the basic resources, information database, schedule implementor, and execution monitor. These components interact with the scheduler to manage the resources. More detailed information on resource management in Legion can be found in [26].

4.3 Summary

Security is perhaps the most important aspect of any metacomputing system. Before organizations are going to be willing to use metacomputing systems, especially in terms of allowing outside use of their resources, they must be confident that they will not be compromised. Users are also concerned with security since they must feel confident that their computations will be secure. This includes the integrity of their data as well as privacy.

The Globus and Legion projects have fundamental philosophical differences concerning security. While the Globus developers preach a need for a rigid definition of what it means for a system to be secure, the Legion project is based on the idea that users should be able to choose between the cost of different levels of security.

The Globus Security Infrastructure focuses on authentication, not authorization. For the most part, authorization is left as a local concern. The GSI is a certificate-based system making use of X509 certificates assigned by the Globus team itself. All GSI algorithms are coded in terms of the Generic Security Services standard, which defines a standard procedure and API for obtaining credentials, authentication, and message oriented encryption and decryption. Authorization is performed locally and users must have local accounts on all systems that they access via Globus.

Legion provides a framework, within which a variety of security mechanisms can be implemented. Based on X509 certificates, Legion objects provide methods for authentication and authorization. Legion provides some simple default implementations of these methods but gives the user free rein to implement customized methods. This allows the users to choose between the high cost of rigid security and the low cost of light security, or anything in between. Legion also provides a mechanism allowingsecurity policies to be enforced externally. This external enforcement can range from very light to very rigid.

Resource management is a complicated issue in a metacomputing environment. One of the goals of metacomputing is to hide this complexity from the user. This is accomplished by each of these systems to varying degrees. Additionally, resource management should afford local administrators full control over their systems.

Globus provides the Globus Security Infrastructure, which focuses on authentication. Authorization is left mostly to the user and local administrator. The GSI is based on mutual authentication and is built on the Generic Security System (GSS). This allows Globus to interface with various security mechanisms such as SSL and Kerberos. This architecture allows Globus to provide a choice between simple security and very rigid security protocols. Since Globus requires users to have local accounts on all resources accessed, it is easy for administrators to protect their resources in the same way they do without running a Globus system.

Globus provides various mechanisms for managing the resources in the system. These include mechanisms for selecting resources as well as mechanisms for allocating resources and starting processes on the allocated resources. Globus also provides interfaces to several schedulers and batch queuing systems, allowing sites to maintain local resource usage policies. However, the user is still required to learn and understand a resource specification language to help the system to select appropriate resources.

Globus provides strong support for sites controlling their resources and for maintaining local policy. Globus sites have complete control over what users have access to what resources. Globus also provides a mapping between a globusid and local userid, giving Globus sites additional methods for controlling user access to resources. This includes quotas and limits on time spent on particular resources. Additionally, Globus security can be built on other security systems, such as Kerberos, giving local sites more control over local security policy. Currently a new version of GSI under development that directly calls the Kerberos API (avoiding the need to use the GSS binding to Kerberos), and so it will be possible to have a Globus system that works only within a Kerberos realm and does not use certificates.

While Legion provides basic mechanisms to support security in a distributed, cross-domain environment, much of the enforcement is left to the users and local administrators, including support for Kerberos authentication if available. Legion provides a means for achieving a level of confidence acceptable to the individual. In this spirit, Legion implements some simple default security policies, but gives users and local administrators the ability to implement other, possibly more stringent, policies, and seamlessly integrate them into a Legion system.

Like Globus, Legion also provides mechanisms for resource management, in the form of collections, schedulers, PCDs and enactors. These mechanisms (and object classes) are fully extensible with Legion providing a number of default implementations, but giving the user and local administrators the ability to substitute custom implementations. Provided with the appropriate scheduler, the user need not supply any additional information to assist the system in resource selection. However, the default schedulers are simple, and will not provide optimal performance. In order to achieve better performance, schedulers must be provided which understand the resources required by the objects to be scheduled usually from within their own class object creation implementations.

Legion also supplies mechanisms by which local administrators can control their systems. This is mostly the configuration of HostObjects, collections, enactors and the Legion Schedulers. These objects allow the local administrator to select resource management and usage policies that meets his needs. This control can range from simple to very complex, depending on the objects used. If there are no existing objects that enforce local policy as needed, administrators have the ability to implement new object classes that will meet their needs. Like Globus, Legion security can be built on top of an existing security mechanism such as Kerberos.

.

5 System Functionality

Even if a system is easy enough to install and maintain, easy enough to learn and use, provides acceptable levels of security, and provides sufficient levels of site autonomy, it is not going to be used if it does not provide sufficient functionality. Areas of functionality of special concern in a metacomputing environment include the file system, language support, and fault tolerance. The most important, perhaps, is performance. If users cannot get at least as good of performance from a new system as they do from existing systems there is reduced motivation to move to the new system unless there is some other greater value or capability added elsewhere. MetaComputing systems do promise extra value in the form of global transparent file access or the ability to run much larger jobs that currently possible under individual MPP systems for example. Thus outright performance of such systems may not be as good as optimized vendor software but it should allow for still reasonable performance if it is to be used for High Performance Computing applications.

5.1 The file system

The metacomputing concept requires access to hosts running in different administrative domains. This cause four main problems in regards to the file system; application binaries may not be present at all sites, applications may not be able to read and write needed files, sharing of data between collaborators is difficult, and access to remote databases is difficult [19]. Two requirements of a metacomputing system are to provide a file system with a single namespace and provide mechanisms for I/O.

5.1.1 Globus

Globus does not present a file system with a single namespace. This is a conscious design decision intended to provide increased performance and simpler implementation. However, Globus does provide mechanisms for automatically staging executables and copying data files. This solves all the above mentioned problems except access to remote databases. Remote database access is not addressed by the Globus system.

Executables can be automatically staged by including the "-stage" flag to the "globus -job-run" command. This causes the executable to be automatically staged for execution, and automatically removed once the execution terminates [13]. This may be necessary if quota limitations on the remote resource preclude storing the executable on a persistent basis. It also allows the maintenance of a single executable with the ability to execute it on multiple hosts, which reduces complexity for the user. The automatic removal policy may cause problems in the case of multiple executions at the same site by performing more copies than required, a situation the caching mechanism in Legion avoids.

Sharing of resources can be accomplished by use of the "globus-url-copy" and "globus -rcp" commands. These commands are used for transferring data between hosts. These commands use the *Global Access to Secondary Storage* (GASS) service [13]. GASS and Globus are cooperating services. That is, GASS makes use of Globus services for security and communication, while Globus uses GASS services for executable staging and real-time remote monitoring [28].

Globus makes use of GASS to provide the user with simplified mechanisms for performing remote I/O. GASS is designed to minimize the number of changes needed to execute existing applications. A user need only change the calls for opening and closing files. GASS provides the following commands for performing these operations: "globus-gass-open", "globus -gass-close", "globus -gass-fopen", and "globus -gass-fclose". These routines take a URL in place of a file name. Only the open and close calls need to be modified in application code. All other I/O calls remain unchanged.

More information on the use of the Globus file system can be found in [13] and [28]. More detailed information on the GASS design and implementation can be found in [28].

5.1.2 Legion

The Legion system focuses on file system support, rather than database support. Legion has constructed a federated file system using the host file systems as component elements. The Legion file system maintains what they call a "context space". A Legion context space is globally named and globally accessible. A file in the Legion file system is identified by a globally recognizable path name, say "/users/person/myfile". A Legion user accesses this file with the same name, regardless of their location. In fact, the physical location of the file is hidden from the user. The file itself may reside anywhere in the system, and may even be physically stored in different locations at different times. All the complexity involved in locating and accessing the file is hidden from the user. This presents a true single namespace view to the user. Although this single name space is only valid for a single Legion Domain, and if multiple Domains are combined, then the non-local domain is prepended to the context names as in "/domain/domain.XXX/users/otherperson/theirfile".

Legion also provides a set of library routines supporting the management of files. When a user opens a file, the Legion system examines the call to determine if the file is a Legion file or a local file. Local file operations are handed to the host operating system. Legion file operations are trapped and handed to an object that handles the file operation. There is also a set of routines for manipulating the Legion namespace. These routines are analogous to the Unix routines mkdir, rm, mv, etc. Files enter the Legion namespace by either being created in the Legion namespace, moved into the Legion namespace, or linked into the Legion namespace [17, 19].

For detailed information on how to import Unix files and directories into the Legion context space, please see [29] and [30].

Before a program can be executed in the Legion system, it must be registered. This is done with the "legion_register_program" or "legion_register_runnable" command, depending on if the program is Legion compatible or Legion incompatible. One of the parameters to these commands is the architecture under which the program runs. Legion uses the "legion_run" command for executing programs on resources remote to the program executable. The user may specify a host on which to run the executable, otherwise Legion will select a resource by random, ensuring it is of the architecture specified during the registration step.

The "legion_run" command is also used to specify input and output files that should be used by the program. This may include either Legion context files or local Unix files. This allows Legion programs to access input and output files that are not co-located with the program execution. More detailed information on executing jobs in the Legion environment can be found in [29] and [30].

5.2 Language support

Language support is important in any high-performance computing system. Since scientists are typically unwilling to learn a new programming language, if a system does not support the language of their choice they simply will not use it. The most common language for scientific code is Fortran, so there is a particular need for any system to support Fortran, including F77 and F90. For a system to be truly successful, however, it should support a wide range of languages, possibly including C, C++, HPF and ADA.

5.2.1 Globus

Globus provides very strong language support. Since Globus does not require any special compilers, any executable that runs on a particular architecture will also run on that architecture under Globus. Thus, by default, Globus has language support for any sequential language. Globus also provides interfaces to a number of parallel programming interfaces. These include MPI, Compositional C++, Fortran M, nPerl, and NexusJava. Since Globus supplies a complete implementation of MPI, it also supports tools layered on top of MPI, such as many HPF systems [1].

5.2.2 Legion

Legion also provides strong language support. Like Globus, Legion does not require programs to be compiled with any specialized compilers (though this is possible). This results in Legion being able to run any executable that can run on a particular architecture. However, high-performance computing is really interested in running parallel programs. Legion originated from work on the Mentat programming language, and thus Legion has supported Mentat from the very beginning. Legion also supplies Legion versions of the MPI and PVM libraries. Legion also provides basic Fortran support for parallelizing Fortran code. This support allows Legion directives to be added to Fortran programs, enabling a Legion compiler to convert the program into a Mentat program. See [31] for details on Legion's Fortran support.

5.3 Fault tolerance

Fault tolerance is important in a large-scale distributed system. This is because in a system as large as those envisioned by metacomputing systems, it is a certainty that at any given moment many hosts, communication links, and/or disks will have failed [17, 19]. This has implications both in terms of the overall system and in terms of individual applications. For a large system, such as those described here, to be used, users must be confident that resources will be available when needed. Application level fault tolerance is also important. Many applications that may make use of these large systems have execution runs that may last for days, or even months. It is important that such applications have a means for detecting and recovering from resource failures. It is unacceptable to have to totally restart such an application after it has already been running for an extended period of time.

This section discusses the fault tolerance mechanisms built into the two systems being evaluated for this report.

5.3.1 Globus

Early experiences with the Globus system helped the Globus developers to identify one serious issue in terms of fault tolerance. This issue involves the failure or unavailability of resources selected for a computation. Early implementations of Globus resource management provided for selecting a set of resources and then sequentially allocating and beginning execution on these resources. The problem that arose was when one of these resources was found to be unavailable, the jobs previously started on other resources had to be terminated, and the whole process of allocation restarted. This was seen as a major deficiency since resource failure was frequent [5, 35].

This deficiency was addressed by the design of the *Globus Architecture for Reservation and Allocation* (GARA). The primary goal of the GARA work was to support advanced reservations and co-allocation in a distributed environment. In this design, reservation and allocation are separated. This results in a resource needing to be reserved before resource allocation. As a result, the system can be confident that all selected resources are available before beginning to start execution on the desired resources. This prevents the problem of a resource being found unavailable after execution has begun on other resources. Detailed information on the design and implementation of the GARA layer can be found in [36].

The Globus developers also recognize the problem of application level fault tolerance. To this end they have designed and implemented the Globus *HeartBeat Monitor* (HBM). The Globus HBM provides an API for client applications. This API provides mechanisms for registering with the HBM either internally or externally. That is, an application can explicitly register with the HBM or another process can cause a process to be registered. The HBM provides a mechanism for monitoring the health of registered processes and reporting failure to the appropriate process [16, 22]. Details on the design and implementation of the Globus HBM can be found in [16].

It is important to note that Globus does not provide application-level fault tolerance. Instead, what it provides is a mechanism for a Globus application to detect application failures. It is expected that Globus applications will use this fault detection mechanism to implement application-specific fault tolerance and response mechanisms.

5.3.2 Legion

The Legion developers recognized early in the design phase that fault tolerance was a key design issue in a metacomputing system [10, 18, 21, 26, 32, 33, 34]. They identified two types of fault tolerance, fault tolerance with the Legion system, and application-level fault tolerance. The Legion team first addressed the issue of fault tolerance within the Legion system. That is, the Legion system itself handles hardware and network faults [17, 19].

The Legion philosophy with respect to fault tolerance is based on two fundamental observations:
(1) fault tolerance algorithms require redundancy in space or in time. Thus, users should be able to select the level of fault tolerance that they need. The cost in terms of resource consumption will be proportional to the level of fault tolerance selected.
(2) fault tolerance algorithms are in general difficult to design and implement correctly. Thus, fault tolerance experts should be the ones designing and implementing algorithms, not programmers.

Based on these observations, Legion provides developers with a reflective model for extending the functionality of objects "Enabling Flexibility in the Legion Run-Time Library". In particular, these extension mechanisms provide fault-tolerance developers with the ability of transparently integrating fault-tolerance techniques into grid applications.

To date, application-level fault tolerance consists of rollback/recovery of MPI SPMD applications and replication for bag-of-tasks applications. Future release of Legion will incorporate additional mechanisms and will enable the transparent integration of message logging and replication techniques for user applications "Using Reflection for Incorporating Fault-Tolerance".

5.4 Performance

From a user's perspective, performance is perhaps the most important factor in judging a large software system. If a system does not provide performance, users will simply not use it unless other features either make it attractive or that it is the only method available to run very large jobs spread across multiple systems and sites etc. This section presents various performance results under MPICH with the ch_p4 device, Globus, and Legion.

These tests were run using a cluster of machines with dual Pentium III 550 MHz processors running the RedHat 6.1 version of Linux. Tests were run with a single process per machine.

Performance Test Notes.

These tests were run on a cluster of machines, not on multiple MPP systems under multiple batch control systems as is envisaged as the true use of MetaComputing Environments such as Legion and Globus. Thus it may be construed that these results are unfair when compared to those obtained from an optimized MPI implementation on a tightly coupled LAN[1]. That is why the authors chose to compare the results of Legion and Globus to the ch_p4 over IP implementation of MPICH, as this most closely mimic the type of interconnection most commonly found between multiple systems at multiple sites.

[1] The Torc cluster at ICL, University of Tennessee, Knoxville, is a multi-interface system supporting simultaneously both fast and gigabit Ethernet as well as a range of Myrinet devices. Multiple tuned MPICH implementations based on both GM and BIP are installed.

The authors of this report believe that the performance of these implementations can be used to gauge how well such implementations will scale when distributed across multiple sites. This is mostly from experience gained with using MPI interconnection software such as MPI_Connect [37] and PACX-MPI [40] which both allow vendor MPI implementation speeds within MPPs and use TCP/IP between MPPs.

In defense of both Globus and Legion, they both allow vendor (or native) MPI jobs to be started ona target system. But how they allow this for a multi-machine heterogeneous distributed single MPI application is unclear. This is discussed briefly in section 5.4.2.

Bandwidth test

The first test is a bandwidth test. This test simply bounces a message back and forth between two processes. This test runs with two processes, each on a different machine. This means there is no shared memory communication. Table 5.4.1 presents the results of running this test with each of the different MPI implementations. Figure 5.4.1a is a graphical representation of this data, and Figure 5.4.1b shows the bandwidth achieved.

By studying these results, we see that Globus has a much higher overhead for non-zero length small messages when compared to MPICH using the p4 device. As messages get larger, Globus performance gets within around 10% that of MPICH/p4. Additionally, we see that Legion is outperformed by the other two implementations. We conclude that Legions object oriented design must introduce considerable communication overhead, which as can be seen is constant for most small message sizes. The step at 800 bytes for Legion was consistent for our measurements. While the Globus and MPICH/p4 performance approaches that expected of standard TCP/IP sockets for large messages, Legion does not, and its large message performance falls behind by around 50% compared to the other systems.

Message size in bytes	MPICH/ch_p4 Time (ms) - Mbytes/sec	MPICH-G Time (ms) - Mbytes/sec	Legion MPI Time (ms) - Mbytes/sec
0	0.178	0.187	2.01
8	0.180 – 0.042	0.416 – 0.018	2.10 – 0.004
80	0.195 – 0.389	0.342 – 0.222	2.07 – 0.037
800	0.279 – 2.733	0.347 – 2.198	2.26 – 0.338
8000	1.03 – 7.406	1.09 – 6.981	37.1 – 0.206
80000	7.47 – 10.212	8.00 – 9.534	17.6 – 4.332
800000	73.4 – 10.388	82.3 – 9.259	178 – 4.280

Table 5.4.1 *Round trip message passing times*

Collective operations

We also tested collective operations under the three systems. These collective operations include broadcast, barrier, and a max reduce operation. Broadcasts were performed for 2, 4, and 8 processors with message sizes of 1 Kbytes and 1 Mbytes. Barrier and reduce operations were also done on 2, 4, and 8 processors. The reduce operations used 128 doubles per processor for one test and 131072 doubles for another test. The results for these tests are presented in Table 5.4.2, and graphically in figures 5.4.2.1-5.

We see from these results that MPICH-G performs slightly worse than MPICH/ch_p4 for these collective operations. Legion MPI, on the other hand, performs significantly worse than MPICH-G for all cases. These results are similar to those for point-to-point messages where we concluded that Legion MPI introduces significant overhead in terms of message passing performance.

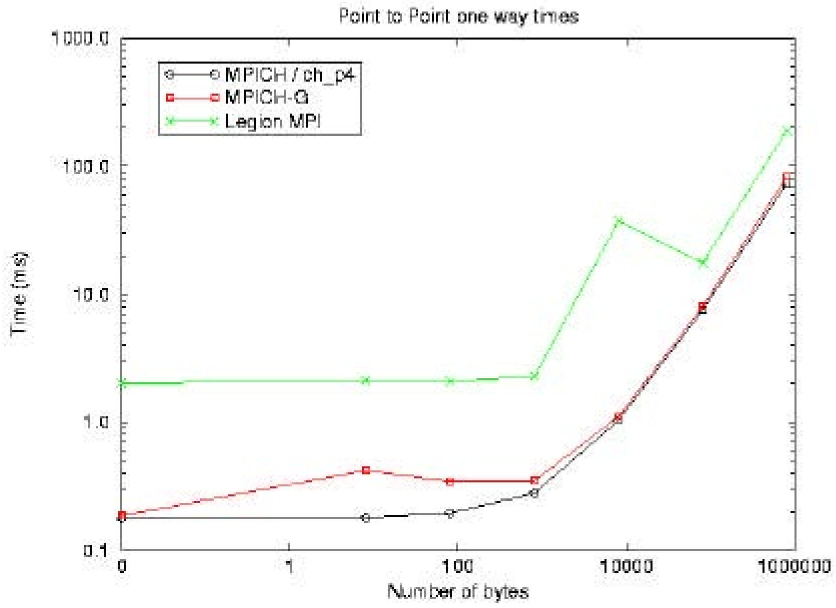

Figure 5.4.1a *one way message passing times*

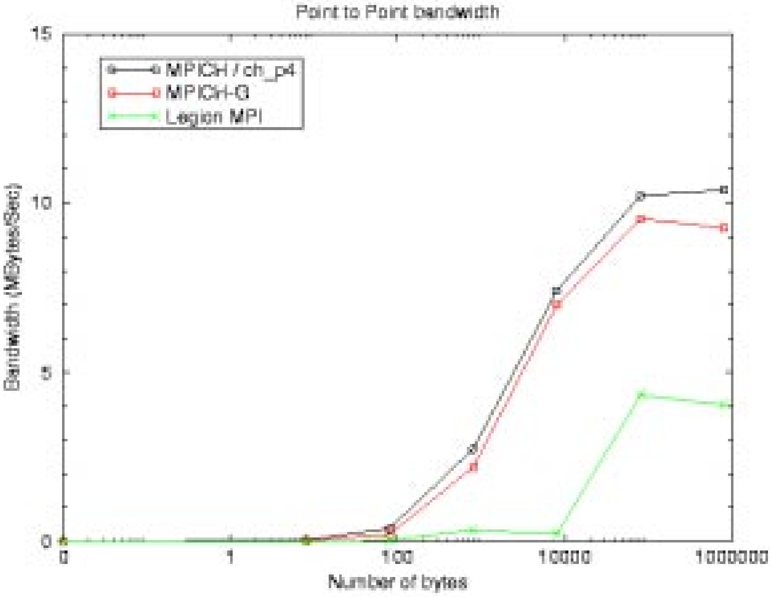

Figure 5.4.1b *bandwidth achieved for various message sizes*

Operation	# Processors	Size (bytes)	MPICH/ch_p4	MPICH-G	Legion MPI
Broadcast	2	1024	0.374 ms	0.426 ms	3.816 ms
Broadcast	2	1048576	97.82 ms	106.1 ms	273.6 ms
Broadcast	4	1024	0.618 ms	0.733 ms	30.23 ms
Broadcast	4	1048576	194.2 ms	309.5 ms	398.2 ms
Broadcast	8	1024	1.022 ms	0.862 ms	49.59 ms
Broadcast	8	1048576	291.4 ms	607.8 ms	785.9 ms
Barrier	2	-----------------	0.210 ms	0.232 ms	4.12 ms
Barrier	4	-----------------	0.405 ms	0.439 ms	7.67 ms
Barrier	8	-----------------	0.741 ms	0.706 ms	16.2 ms
Reduce	2	128 doubles	0.123 ms	0.153 ms	23.34 ms
Reduce	2	131072doubles	117.0 ms	134.1 ms	171.4 ms
Reduce	4	128 doubles	0.237 ms	0.299 ms	30.96 ms
Reduce	4	131072doubles	224.9 ms	232.1 ms	298.1 ms
Reduce	8	128 doubles	0.346 ms	0.532 ms	43.80 ms
Reduce	8	131072doubles	333.1 ms	320.8 ms	645.0 ms

Table 5.4.2 *times for collective operations*

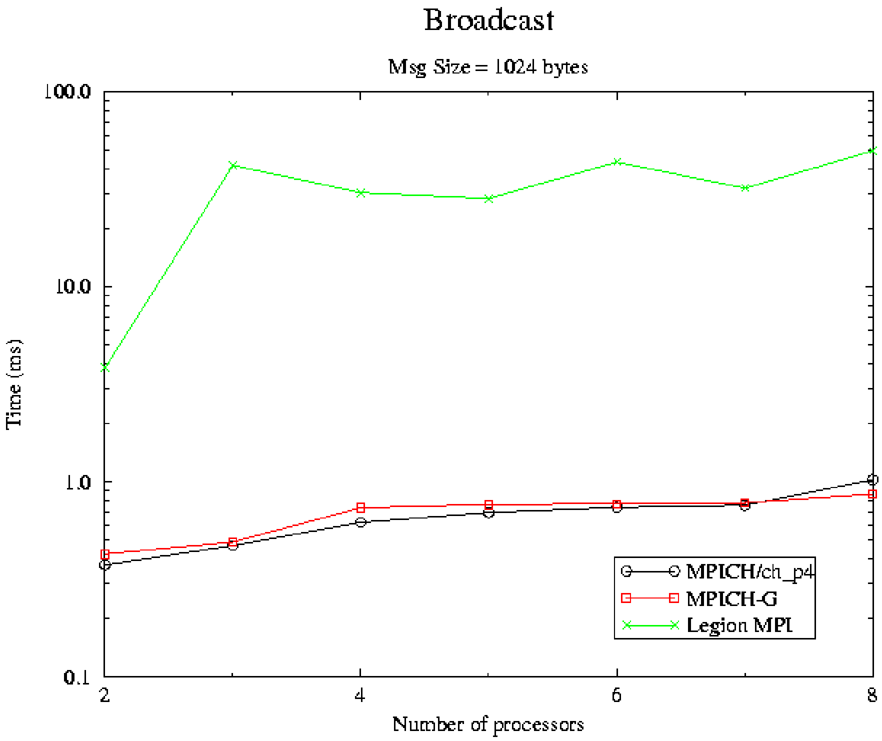

Figure 5.4.2.1 *times for small broadcast message*

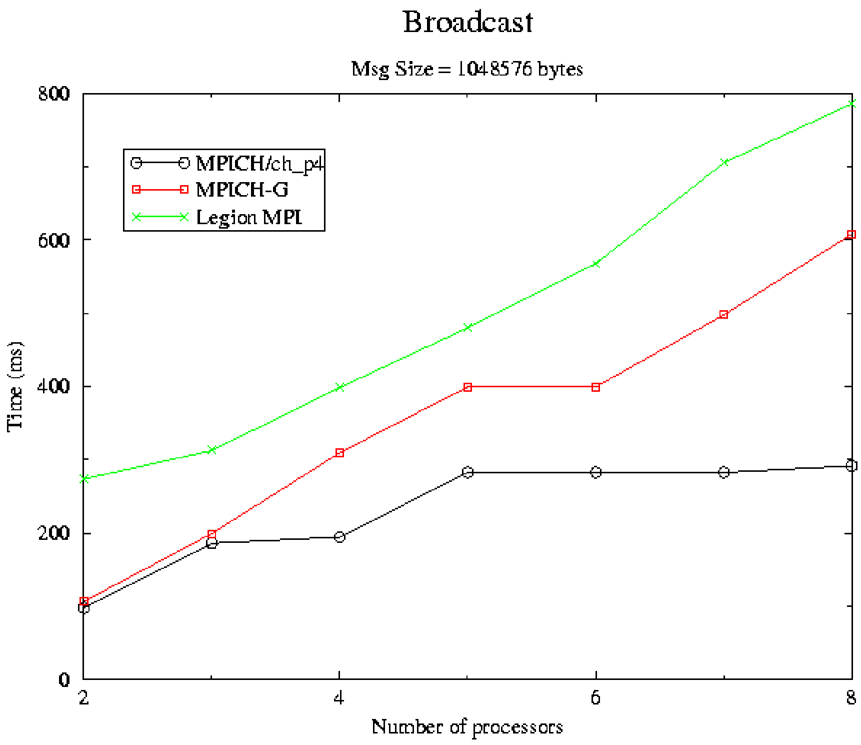

Figure 5.4.2.2 *times for large message broadcast*

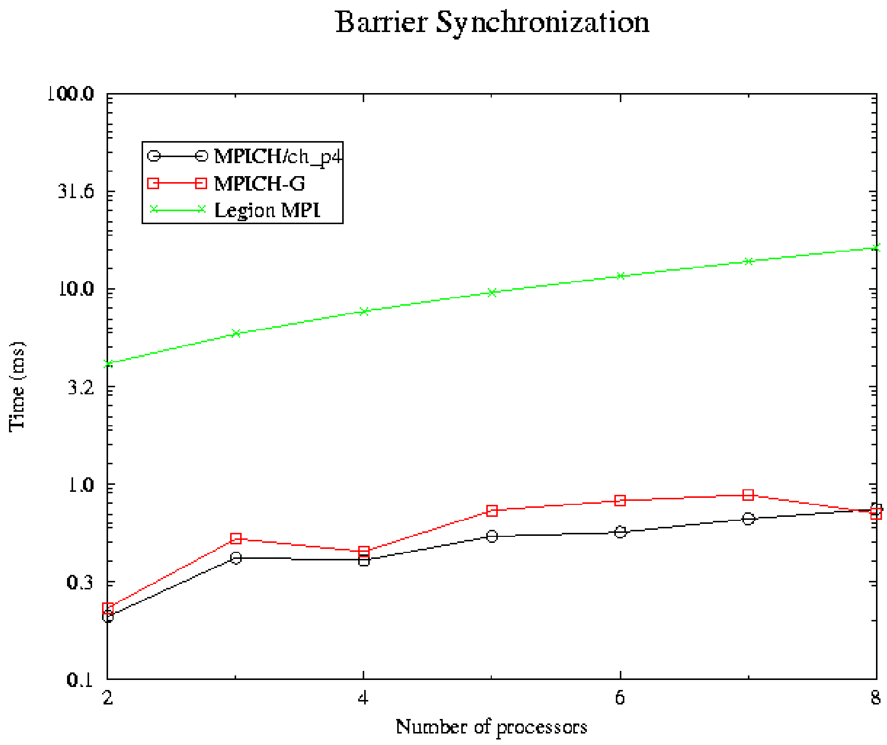

Figure 5.4.2.3 *times for barrier operation*

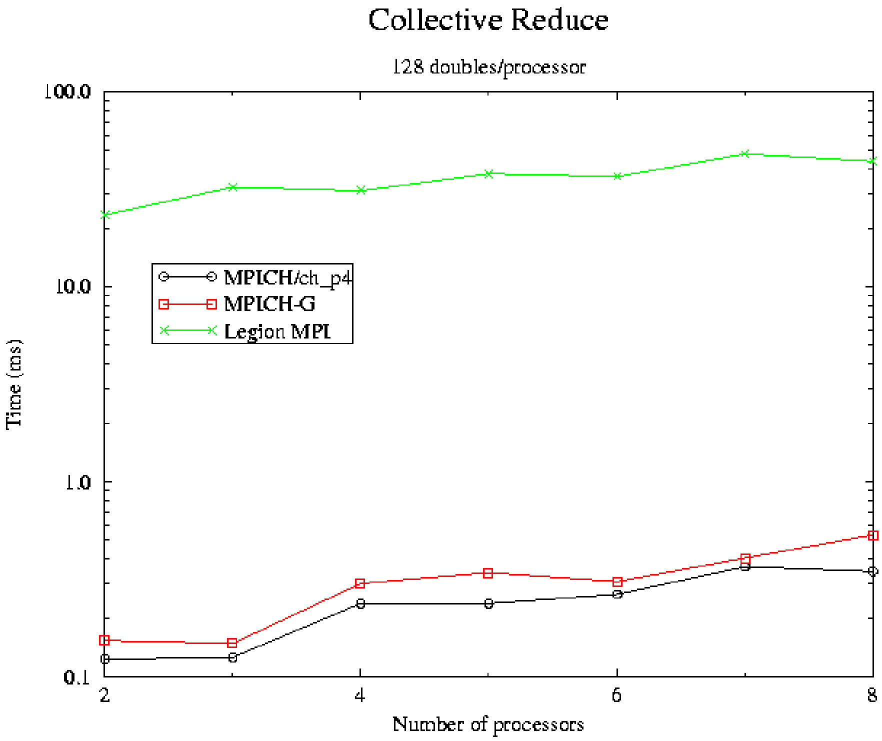

Figure 5.4.2.4 *times for small collective reduce operation*

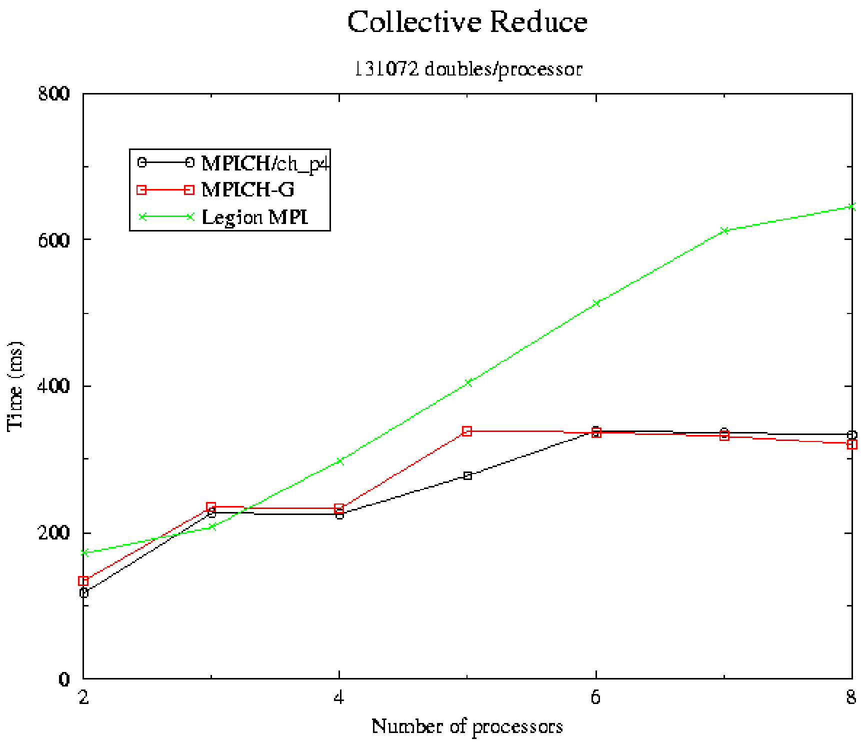

Figure 5.4.2.5 *times for large collective reduce operation*

LU factorization

Another test we ran solves the equation Ax=B using LU factorization with row partial pivoting. This test program comes from the ScaLAPACK numerical library. For larger matrices, this test is dominated by computation, with little communication cost relative to computation. Table 5.4.3 presents the results of running this test program on 4 processors using a 2x2 processor grid and a block factor of 40. This data is also shown graphically in figure 5.4.3.

Size of matrix A	MPICH/ch_p4 Time (s) - Mflops	MPICH-G Time (s) – Mflops	Legion MPI Time (s) - Mflops
1000 x 1000	2.39 – 255.50	3.58 – 168.11	
2000 x 2000	9.52 – 543.61	18.20 – 280.92	
4000 x 4000	50.14 – 840.04	115.74 – 363.68	
5000 x 5000	89.25 – 925.31	216.37 – 381.67	
6000 x 6000	145.84 – 980.68	363.45 – 393.65	

Table 5.4.3 *LU factorization on a 2 x 2 processor grid*

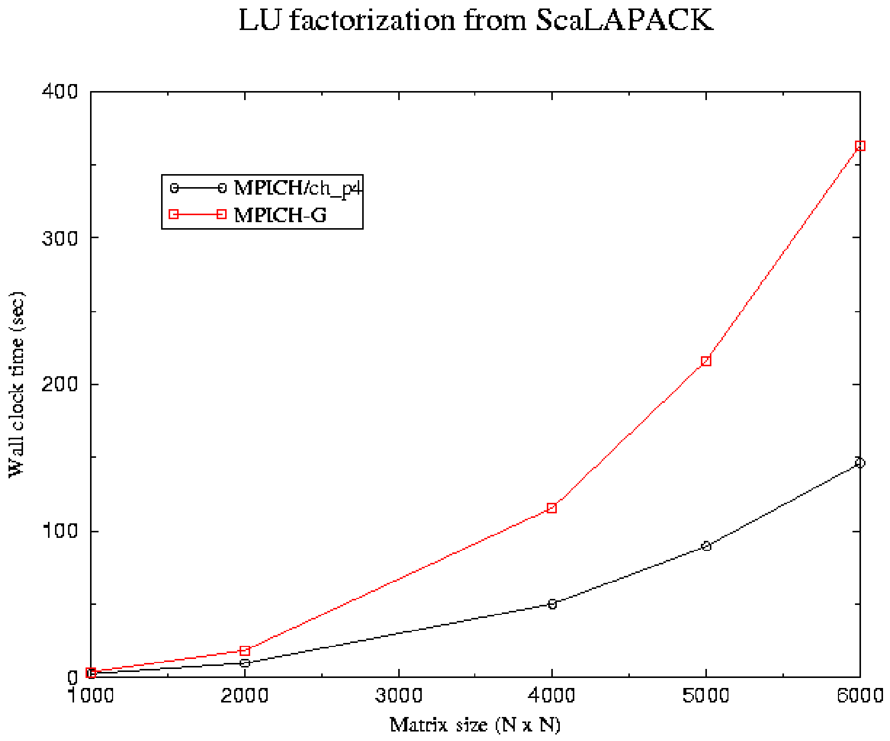

Figure 5.4.3 *LU factorization on a 2 x 2 processor grid*

As can be seen from the results, this test runs considerably slower under Globus than it does using MPICH-ch_p4. Since this application is dominated by computation, as opposed to communication, we must conclude that the use of Globus contributes significant overhead. We initially believed this overhead is introduced by the Globus job-manager (which runs on all processors running a Globus application) stealing CPU cycles from the application, but we later disproved this as is discussed in section YYYY .

Due to hardware problems as we were completing work for this report, we were unable to get consistent results for LU factorization running under Legion.This is not seen as a significant problem as we were able to run the Linpack benchmark as described below.

Linpack

The final application test we ran is based on Linpack and solves a general linear system. We note that for these tests, smaller matrices lead to the execution being dominated by communication, while larger matrices lead to the execution being dominated by computation. These tests were run on 4 processors with a 2x2 process grid and a block factor of 40. Table 5.4.4 and figure 5.4.4 represents these results.

These results are consistent with those presented above. We note that MPICH/ch_p4 outperforms both MPICH-G and Legion MPI. More interesting is a comparison between MPICH-G and Legion MPI. As noted above, this test is dominated by communication cost for smaller matrices. This explains why MPICH-G outperforms Legion MPI in the first couple of rows. Since Legion MPI has a lower peak bandwidth for large message communication, we expect MPICH-G to perform better for the larger matrix sizes. However, for larger matrices, Legion MPI outperforms MPICH-G. These executions are dominated by computation. The poor relative performance of MPICH-G for these executions was some cause for concern and was investigated as shown below in section 5.4.1.

Size of Matrix	MPICH/ch_p4 Time (s) - Mflops	MPICH-G Time (s) – Mflops	Legion MPI Time (s) - Mflops
1000 x 1000	1.47 – 455.2	2.49 – 268.4	13.30 – 50.23
2000 x 2000	7.09 – 753.2	15.45 – 345.6	33.19 – 160.9
4000 x 4000	42.56 – 1003	107.45 – 397.3	98.66 – 432.7
5000 x 5000	77.97 – 1069	202.98 – 410.7	151.06 – 551.9
6000 x 6000	128.79 - 1119	343.36 – 419.5	214.68 – 671.0
8000 x 8000	288.30 -- 1184	789.87 -- 432.1	415.56 – 821.6
10000 x 10000	546.30 -- 1221	1529.4 – 436.0	722.75 – 922.6

Table 5.4.4 *Linpack on a 2x2 processor grid with block size 40*

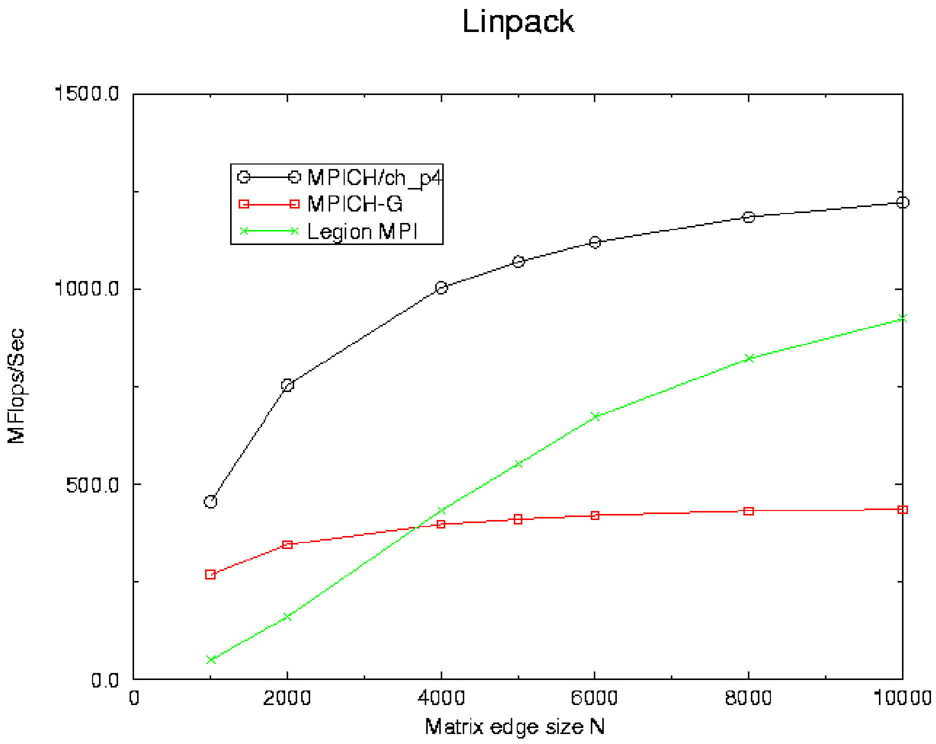

Figure 5.4.4 *Linpack on a 2x2 processor grid with block size 40*

One might expect Legion MPI and MPICH/ch_p4 to perform similarly for executions dominated by computation. However, the communication performance of Legion MPI is less optimal than that for MPICH/ch_p4 on a local network and thus it only achieved 75% the peak performance of MPICH on this test. Section 5.4.2 discusses this point further.

5.4.1 Investigating Globus MPI performance

The question arose as to how a point to point and collective operation bandwidth test suite could give good results but actual multi-process applications did not. This was investigated by the authors in an attempt to locate if this was a computation problem as in Globus was cycle stealing via the Globus daemons or rather a feature of the communication software.

Experimentation was performed by developing a completely controllable, instrumented application. The application took the form of a ring of processes which compute for a configurable time, and then all pass a variable length message around the ring and then compute again and so on. The timing within the application was switch-able from MPI_Wtime() to gettimeofday() to avoid any synchronization issue that could arise with the MPI call.
The results of the tests were dumped to file so individual messages could be examined to check for rouge messages that take excessive time but do not show up in averages. Also MPE clog files were produced using the –mpilog MPICC option and then examined using the MPICH Jumpshot application.

Tests of Legion and MPICH/ch_p4 appeared similar and are shown in figure 5.4.5. But the tests for Globus depended very much on the both the computation time and message sizes. In the worst case individual Globus message passing times defaulted to the same time as computation step.

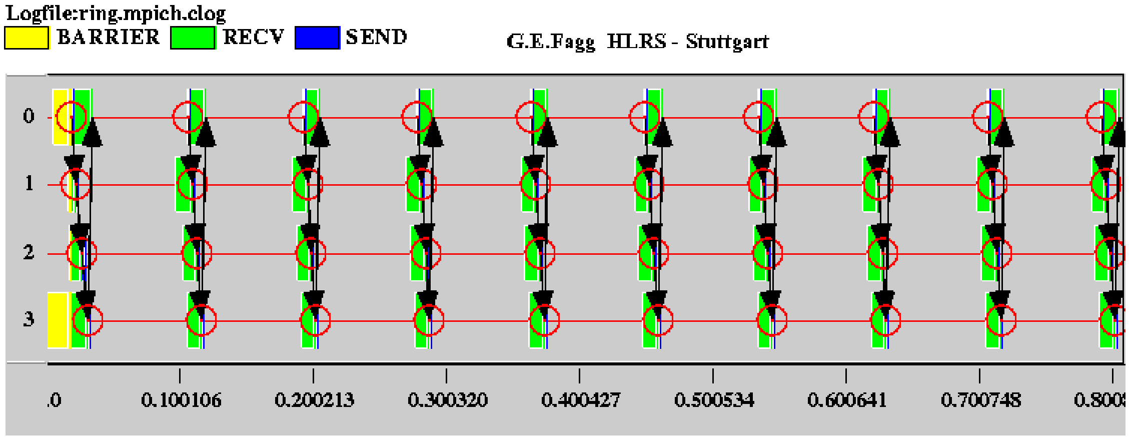

Figure 5.4.5 MPICH/ch_p4 execution of a test ring application.

Figure 5.4.6 shows Globus for the same application test with but using only small messages. This result is almost identical to the MPICH version.

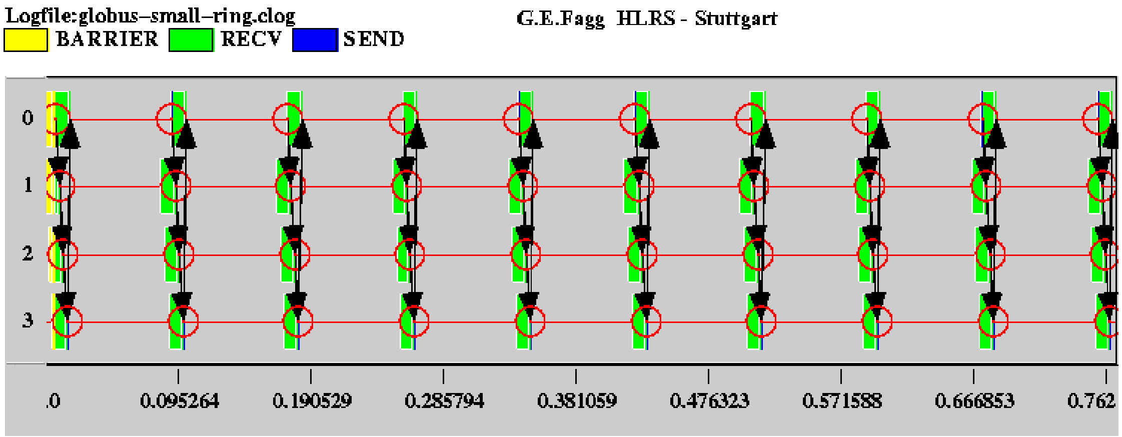

Figure 5.4.6 Globus execution with small messages within a ring.

Figure 5.4.7 Shows what happens to the Globus execution of the ring with larger messages. Figure 5.4.8 gives this in greater detail zoomed in image.

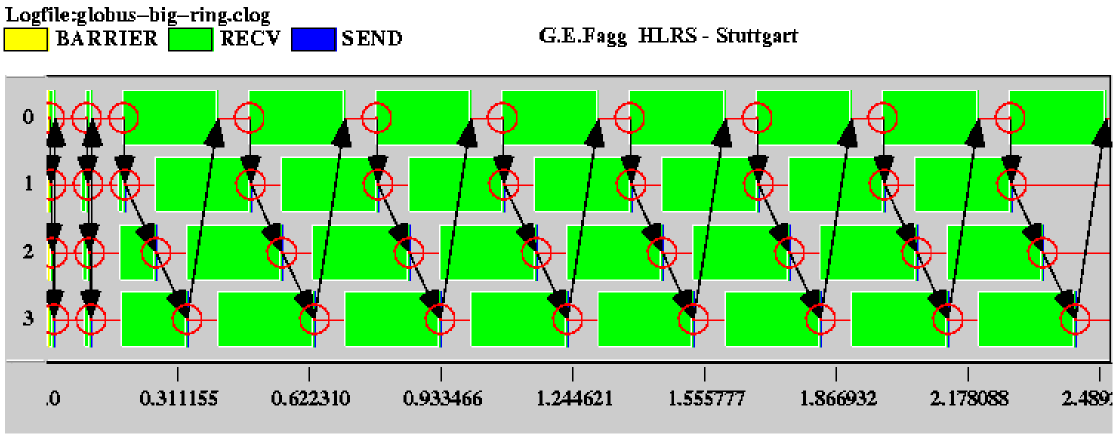

Figure 5.4.7 Globus ring execution for larger messages.

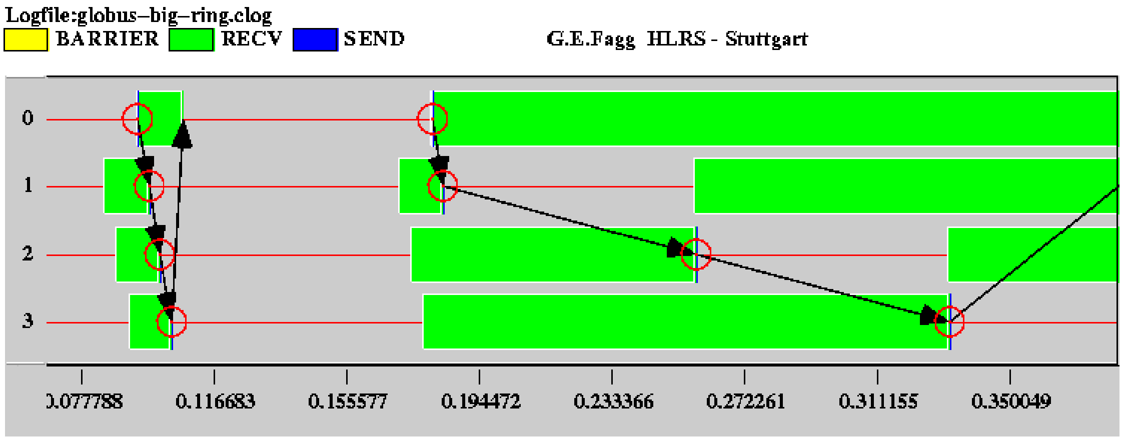

Figure 5.4.8 Globus execution in detail for larger message ring.

The difference in message size required to trigger these events was from 3000 to 4000 doubles or as the message size (with internal headers etc) approached a 32Kbyte limit. What appears to be happening is that during a send-receive pair of operations, if not all the message can be sent immediately, the sending process waits until the next MPI call to complete the operation (i.e. does not make progress between calls as it is defined in the MPI specification). Thus if the receive starts too late and the sender has already continued into some computation, the receive operation will not complete until the sender reaches another MPI call. An effect similar to this is when pairs of communicators always send additional acknowledgements messages to let each other known when to free message buffers. It is worth noting that we tested Globus with both the single and multithread versions of its MPI device library.

The overall effect of this is to serialize the complete computation, i.e. only a single process is active at once depending on the communication pattern used. Thus returning to our HPC test, the flattening of the Globus performance relates to achieving almost peak performance on a pair processes, depending on the shape of the computation. In this case a 2x2 thus the expected performance would probably be more than twice 436 Mflops, putting it in the 870+ Mflop range.

We believe this problem to be a fixable bug, but we were advised against changing to MPICH 1.2.0 with the Globus device due to possible compatibility problems. So as yet we do not know if this problem as been rectified.

5.4.2 MetaComputing with MPI

We attempted to run MPI jobs across multiple sites using both Legion and Globus. We were unable to test Globus in this way as we did not have user account at other sites where Globus was installed. Our problems with running Legion were different, and caused by firewall restrictions between different sites, including Oak Ridge National Laboratory and the Computer Center (HRLS) at the University of Stuttgart Germany. The former site restricted IP ports and the later restricted ports as well as enforced strict access via SSH only.

We were however able to merge two separately running Legion Domains and allow them to share their context spaces using the legion_combine_domains call. So we understand that by creating a hostfile of the following structure that a multi-site MPI application run is possible:

```
/hosts/machine1 1
/hosts/machine2 1
/domain/domain.XXX/hosts/machine3 1
/domain/domain.XXX/hosts/machine4 1
```

Both the Legion and Globus teams have informed us that they regularly perform Meta-Computing multi-machine and multi-site runs for a number of projects. These projects are listed in Appendices C and D. Currently we are unsure as to how many of these are continuously running projects, which regularly execute over multiple sites or are just demonstrations or examples used for feasibility studies and HPC challenges.

5.5 Summary

Globus provides the user with mechanisms for automatically staging executables, copying data files to and from remote hosts, and performing remote I/O. This is all handled by the GASS server, which supplies a simple interface and requires very little code modification on the part of the user. While Globus does not attempt to implement a full file system, features important to the targeted users (the high-performance community) are present.

Legion provides full file system support and presents the user with a view of a single namespace. Legion users can access files from anywhere in the system, with no knowledge of the physical location of the file. This includes a full API for opening, closing, and manipulating files. This does require significant code modification for file access. This also requires the user to register executables so the system will be able to locate them when they are to be run. Finally, Legion provides for specifying input and output at the command line, giving executables access to remote files.

Both systems provide strong language support. Globus includes full implementations of interfaces for MPI, Compositional C++, Fortran M, nPerl, and Ne xusJava. The inclusion of a complete MPI implementation provides support for many HPF systems. Legion provides interfaces to MPI and PVM as well as support for the Mentat parallel programming language. Additionally, Legion provides support for converting Fortran programs into Mentat via programmer included Legion directives.

Both Globus and Legion provide some degree of system-level fault tolerance. Globus supports fault tolerance in the resource allocation stage, allowing the system to select only available resources for allocation. Globus does not, however, support recovery from resource failures once the computation has begun. While Globus does not support application-level fault tolerance, it does provide a mechanism for fault detection. The Globus HeartBeat Monitor provides a means for applications to detect faults and react accordingly. Legion supports system-level fault tolerance both prior to resource allocation and after execution has begun. Legion removes failed resources from its system view and also restarts objects executing on a resource if that resource fails.

While Globus seems to have comparable message passing performance in relation to MPICH/ch_p4, there is a bug that affects real computational applications. Legion, on the other hand, does not provide the same level of performance in message passing, at least for local area networks. This is probably due to its object based design. However, it is possible to use either Globus or Legion simply for resource management and use native implementations of MPI on MPPs. In fact, this is the way both these systems seem to be being used in several current test environments. That is to say, the metacomputing system is used only to allocate a MPP and MPI jobs are executed using the native MPI implementation. This means that a primary use of metacomputing systems is not being taken advantage of regularly.

6 Future Growth

A final criteria important when deciding whether or not to deploy a large software system is the potential for future growth. In terms of metacomputing systems, future growth pertains mostly to scalability and extensibility.

6.1 Scalability

Scalability in metacomputing systems refers to the ability to add physical resources and participating sites without affecting the performance of the systems. This includes access to the system as well as the performance of applications using the system. The "Distributed System Principle" states: The number of requests to any particular system component must not be an increasing fun ction of the number of hosts in the system [14].

6.1.1 Globus

The Globus developers recognize the need for a metacomputing system to be scalable. To that end they have made efforts to build scalability into the system. One deficiency to this end comes from the fact that users must have an account on every machine that they may access. This is unacceptable if the system is ever going to grow to hundreds of site with possibly thousands of users. Not only are users going to be unwilling to go through th e effort of requesting an account at hundreds of sites, administrators are going to be unwilling to maintain thousands of users on their systems. Not only would this open up vast security holes, but it would require an unacceptably large amount of time on the part of the administrator. This is only a problem, however, if the dream of hundreds of sites being connected is to be realized. For users who wish to use Globus "in house", this does not present a problem. This is supported by the fact that Globus has been shown to be reasonably efficient for deployments consisting of only tens of sites [5].

For Globus to be truly scalable, however, the MDS must be scalable, since it is a centralized point of contact. This is the reason that Globus supports a decentralized directory service. This is essential to scalability. By allowing individual locations to maintain their own MDS tree, traffic to the centralized MDS located at Argonne is reduced. This is accomplished by allowing local sites to contact the centralized MDS and then serve local users [15].

Globus developers have also built scalability into their fault detection mechanism. The Globus HeartBeat Monitor is an unreliable fault detection mechanism, which aids in scalability. This is accomplished by using unreliable UDP, which has the added benefit of lower overhead when compared to TCP [16]. The only concern with using UDP is that some sites, for security reasons, block all UDP packets by default and so, special arrangements would have to be made to allow the Globus HBM to work.

6.1.2 Legion

Scalability was one of the key design objectives of Legion from the very conception of the project [17, 18, 20]. This has resulted in a design that has a certain amount of scalability built in. While there are a few single logical Legion objects, Legion makes use of heavy caching and a hierarchical organization of lower level objects. This leads to limited access to the single logical objects, increasing scalability. Additionally, Legion objects can be replicated to further reduce contention. All of this leads to the conclusion that an increase in the number of Legion computing resources will not impact contention for the few centralized Legion objects [14].

Legion also makes use of the concept of domains with local collections to increase scalability. A Legion installation at a particular site may constitute a domain. Local objects such as collections, schedulers and enactors are responsible for managing domains. This means that no single object is responsible for managing the entire Legion system. That is, control is completely decentralized [14].

The Legion developers claim that Legion is fully scalable, accepting two assumptions about the Legion system. One assumption is that most accesses will be local. That is, the majority of method invocations will be on objects local to the caller. If this assumption is not held then the scalability of the system is dependant on the scalability of the inter-connect network, which is not scalable. The second assumption is that class objects will not migrate frequently, and further, they will tend to stay active for a long period of time relative to instance objects [14]. It seems that these are pretty strong assumptions.

Legion attempts to implement a scalable distributed file system. A federated file system is used, which uses the local host file system as component objects. Initial implementations of this file system were not scalable[19] but this has been rectified in all current releases and performance due to caching is now good.

6.2 Extensibility

“No single policy or set of policies will satisfy every user, so users must be allowed to determine their own priorities and implement their own solutions as much as possible” [21].

Extensibility refers to the ability of a system to add functionality as the system matures and as new needs and protocols arise. Since it is impossible to predict what functionality and protocols will be required in the future, it is important for any system that is to survive long term to have the ability to adapt to changing needs. Additionally, even today different users have different needs. For this reason, it is a mistake for systems to force specific policies on its users. Users need the ability to select what policiesshould be used.

6.2.1 Globus

Globus has several characteristics that aid in its extensibility. For starters, the directory service (MDS) is built upon LDAP, which has been shown to be extensible. This is combined with an extensible Resource Specification Language (RSL) based on LDAP and the Globus MDS. This RSL uses a set of parameter-name terminal symbols which is itself extensible [15, 2]. This will allow Globus resource management to adapt to changing needs.

In addition to this extensible resource management architecture, Globus has an incomplete data model. This allows users to incorporate additional information as new needs arise [15].

6.2.2 Legion

One of the main design objectives of the Legion project was to provide an extensible system [17, 18, 20, 21]. Legion supports this philosophy by providing the mechanisms for system-level services such as object creation, naming, binding, and migration, and by not mandating these services' policies or implementations [21]. This philosophy has resulted in a system that is extensible by its very nature and design. The system consists of a layered design and implementation with a standard mechanism of interlayer communication [18]. Additionally, Legion specifies functionality, not implementation of the system's core objects [14]. This architecture makes it easy to replace system components while maintaining a consistent interface.

Legion's file system is built upon the *Extensible File System* (ELFS) which allows user specification of caching and pre-fetch strategies. ELFS provides the following file abstractions, asynchronous I/O, multiple outstanding I/O requests, and data format heterogeneity. These file abstractions are structured as a user-extensible class hierarchy [17].

The Legion object core consists of extensible, replaceable objects. The Legion event mechanism is a good example of this. Event handlers can be added to or replaced, allowing user applications to respond to events in a manner appropriate to the specific application [18].

The Legion security mechanism discussed above provides another good example of Legion's inherent flexibility. The user is allowed to choose between speed and functionality. While some users may opt to sacrifice speed for additional security, other users may elect to forgo strict security for the benefit of better performance [20]. Even so, in the lowest level of security all credentials are still encrypted and never passed in the plain.

6.3 Summary

The ability for a large software system to evolve and grow is of utmost importance if the system is to survive, as user's needs change, and the use of the system changes. For metacomputing systems, scalability is an important factor in a system's ability to survive. If a metacomputing system cannot grow in size as more people turn to metacomputing, it will never survive.

Scalability seems to be the most difficult of these problems to solve. While both systems possess aspects that address scalability, none seems to have conquered this problem. Both systems still has serious deficiencies in terms of scalability.

Globus suffers from the fact that users need accounts on every machine they are to use in the system. Additionally, the MDS located at Argonne presents a potential bottleneck if the number of users and sites grow to very large numbers. Although this is being offset by the current practice of sites running their own MDS and potentially their own CAs in the future.

In rating the extensibility of the two systems, Legion appears to be ahead of Globus at least in terms of system design. The Legion design philosophy centers on the idea that users should be able to select levels of functionality. This has been accomplished by providing an architecture that lends itself to users replacing key components. Under Globus individual users have far less control over the system components that they utilize.

7 Conclusions

This section presents conclusions about the two systems evaluated in this report. These conclusions are based on our experiences with the systems as well as experiences of others. At the end, we provide a basic summary of the state of MetaComputing, as we see it.

7.1 Globus

Globus is a rapidly maturing metacomputing system being developed at Argonne National Laboratory. While Globus was found to be relatively easy to install and maintain, this was provided the system was maintained as part of the overall Globus system. This requires getting security certificates from the Globus organization and having resources registered with and publicly viewable from the main Globus site. Efforts to set up an isolated Globus system were a general failure, with little assistance being offered by the Globus development team. Once the system was set up, however, few problems were encountered. Using Globus to execute MPI programs was found to be straightforward, with no code alteration necessary, provided no I/O was being performed.

Security and site autonomy is very strong in Globus. Globus security is built on the concept that a system must define precisely what is means for a resource to be secure. Globus provides a X509 certificate based infrastructure focusing on mutual authentication. Trusting the authentication mechanism, local sites are largely responsibly for authorization concerns. Users must have local accounts on all systems that they access via Globus. Additionally, a user must be entered in a Globus mapfile on a resource to be able to use the resource. The Globus Resource Allocation Manager (GRAM) uses this mapfile to determine if a user has permission to use a resource. Not only does this provide authorization to the resources, it grants full site autonomy to the participating sites. Local administrators have full control over what users have access to what resources. The local account also allows local administrators to enforce local policy such as quotas. The GRAM also has interfaces to local scheduler or batch queuing systems, providing local administrator with additional control over their resources. This removes much of the complexity of resource management from the user, making it easier to access multiple, possibly distributed, resources.

The GRAM and Globus Access to Secondary Storage (GASS) services simplify the job of remote execution for the user. While the GRAM handles resource allocation, GASS allows the user to automatically stage executables, copy data files to and from remote resources, and perform remote I/O. Remote I/O can be performed with very little code modification.

Globus provides strong language support and reasonable fault tolerance. Globus has implemented interfaces to MPI, Compositional C++, Fortran M, nPerl, and NexusJava. Providing a MPI implementation allows it to support tools built on MPI, such as many HPF systems. While Globus does not provide true fault tolerance, it does provide strong support for fault detection. Use of the Globus HeartBeat Monitor allows applications to detect component faults and users can implement their own recovery scheme.

The Globus implementation of MPI is known as MPICH-G. The performance of MPICH-G is less than to native MPI implementations, but this is to be expected. The real problem with using MPICH-G currently however is a bug that causes send/receive operations to delayed by not allowing message passing progress while out of an MPI call. This currently introduces significant overheads into computational applications. This will hopefully be rectified shortly by the Globus team. Some Globus users who are running MPI codes are simply using Globus for resource management and using native MPI for their actual applications and so are unaffected.

7.2 Legion

Legion is more of a metacomputing framework, and is being developed at the University of Virginia. While the core Legion system provides full functionality, most default implementations are simple and not very effective.

Legion is simple to install from both pre-compiled binaries and source code. Installation consists simply of untarring the binaries and or sources and typing make. If using pre -compiled binaries it may be necessary to also install the same compiler used to compile the pre-built binaries. This is due to some incompatibilities between different versions of compilers that cause difficulties at link time. Once the binaries are in place, Legion set up and maintenance is quite simple. Difficulty in validating that the Legion install is in it-self correct was a problem that we only solved with direct help from the Legion development team. If Legion is to prosper a self-validation/proving suite of some kind is required. Most problems we experienced have now been covered by changes to the administration guide for Legion.

One significant maintenance concern is the need to completely re-install the system if Legion bootstrap host terminates ungracefully, although more experienced Legion users can 'repair' the context space and OPR directories from snapshot tar files made after installation.

The day-to-day use of Legion for the average user is quite simple once he understands that he has two environments with which to work. The swapping from Legion context space and the host-operating environment such as Unix can be confusing at first but is easy to get accustomed to. Unlike Globus, which provides run-time support tools to hide remote access etc, under Legion this is all taken care of once the application and data are within context space. Other than access to Legion context space via the "legion_" scripts being noticeably slower than the local hosts Unix calls, Legion works well for a user.

Security and site autonomy are open concerns with Legion. This is by design. The Legion developers feel that users should be able to choose levels of functionality. For this reason, Legion comes with very simple default implementations of security and resource management mechanisms. It is expected that third party developers will develop more complicated mechanisms. Legion provides easy ways for these more complicated mechanisms to be added to the system. The major problem with this approach is, very few people have taken the time to develop such mechanisms. In order for Legion to be successful, there must be development of more stringent security and resource management mechanisms. Having said that, Legion support for Kerberos does exist.

One of Legion's strong po ints, in terms of users, is its file system. Legion presents a file system with a single namespace, allowing users to access remote files with no knowledge of where the files are physically located. There is also an interface for file creation and manipu lation. From a user's point of view, a distributed execution need look no different than a local execution. Input and output files can even be specified on the command line, enabling applications to access these files even if they are not co-located with the executing binary.

Legion provides fairly strong language support and adequate fault tolerance. There is a provided interface to MPI and PVM, and Legion provides basic Fortran support. While there is no support for application level fault tolerance, Legion does have strong system level fault tolerance. Failed components are removed from the system view and any objects executing on a failed resource are restarted on other available resources. There are plans to provide some level of application level fault tolerance in the future.

Our experiences show the weakest aspect of Legion is its message passing performance. Legion MPI demonstrates high overheads, although it did outperform our broken Globus MPI under certain conditions. As with Globus, Legion can be used to execute programs using native MPI on a MPP. In fact, the developers suggest this for MPI programs targeted towards single MPP execution.

Legion was designed with scalability and extensibility as key issues. This has led to a system that is very strong in these areas. In fact, Legion is more of a framework than a fully functional system. From the very beginning the developers envisioned others adding functionality to the system, and this ability is built right into Legion. This should make it easy for Legion to continue to evolve and grow. This is dependent, however, on others developing sophisticated components. It is not clear that this is going to happen.

7.3 Summary for DoD MSRC Users

This summary is based on how either of these Metacomputing systems could and might be used at DoD MSRC sites.

Both Legion and Globus alter the everyday running of systems in a direct manner. The real question to ask, is why use Globus or Legion and what would the benefits be? To answer this, we have to remember what already exists at the time of writing at the MSRC sites. This includes:

(1) Across site authentication via across realm Kerberos tickets. I.e. logon to your home site and then transparently authenticate with any other MSRC machine on which you have a valid *account*.

(2) The Meta-queuing system that allows MPI jobs to be scheduled at either the ASC or ERDC MSRC, i.e. across site scheduling.

Both Legion and Globus provide the first feature, in fact if they were adopted by the MSRCs they would use the current in place Kerberos infrastructure, although additional effort in adding Globus gridmap resource/account files and Legion host/user objects would be required. In fact Legion applications are currently run across shared NAVO and ARL MSRCs using Kerberos authentication.

The second issue of across site scheduling is a different matter. Both Legion and Globus can provide very efficient schedules that then filter down to the individual site and machine queuing systems. Thus both Legion and Globus may be better suited to running jobs across the MSRCs than a “Global” version of NASA's PBS system for example. In fact Legion program graph analysis is very powerful as is the Globus reservation and allocation system. The only question to ask here is what happened to the notion of running a single parallel application across multiple systems at multiple sites rather than choosing which individual MPP system is the best to use, i.e. true MetaComputing. NEXUS, and thus MPICH-G was clearly designed to fulfill this need but it is unclear if this is happening every day in *production* environments. Other tools that are much lighter already exist such as PACX[40], Meta -MPI and MPI_Connect [37] that can perform this functionality without affecting the internal performance of each MPP, unlike MPICH-G.

The one issue that is not currently supported across the MSRCs, but is very well supported in different ways by both Legion and Globus is that of global file access. Each user at each site must maintain multiple file systems at the different sites. If a user submits a job to the meta-queuing system they must ensure that no matter where the job runs that it has access to any required data files. Once the job has finished the user is again responsible for collecting these result files from where ever they are stored. Using either Legion or Globus would simplify this, but with different tradeoffs and costs. Both require changes to the application source code, which is not always possible or available. Users of the Globus GASS system, where the application passes data from application to application in a pipeline fashion, have found that the ‘available to the next only when the current has closed the file’model that it uses affects performance, especially as it moves the data back to the original site before allowing the next to re-open it. This lack of logistical scheduling is less of a problem with the Legion system, which can allow multiple copies of data files (objects) in distributed vaults and has extensive caching built in to improve performance. These advantages are offset by the need for a user to move data files via importing or exporting them into Context space as well as the source code changes required.

The bottom line is that both systems offer benefits in terms of file/data management, and some extra job scheduling functionality. This alone may be enough motivation to adopt one system or the other, but they both come with additional costs. Globus needs a lot of system administration support, but the changes to the user are less obvious. Legion needs less system administration support, but the user has to learn to live with the context space model that is not difficult to adjust to. It will be more advantageous when the two systems provide better support for multiple platforms, single job processing.

Which system the MSRCs should adopt depends on which system they feel meets their needs better. The purpose of this report was to review the systems, not recommend one. To this end, Legion is already being used and reviewed by the ARL and NAVO MSRC sites and Globus is currently in use at numerous sites, including a number of national laboratories, NPACI, NCSA and NASA sites. The reports of these individual sites are the best indication as to when either Legion or Globus, if *ever*, is truly ready for prime time on 24 by 7, 365 days a year, production systems as is required by the MSRCs.

Acknowledgements

We would like to thank Marty Humphrey and Norman Beekwilder, both from the University of Virginia Legion team for their assistance in fixing our broken Legion install and *then* fixing the bugs in their MPI implementation that we discovered for them. Thanks also to Steve Fitzgerald for assisting Brett Ellis in setting up our own local Globus MDS system, and Douglas Engert for all the multiple Globus certificates he supplied us. We would also like to thank Matthias Hess and Edgar Gabriel from HLRS, University of Stuttgart for their invaluable assistance with finding the Globus MPI performance problem.
Lastly, thanks goes to Susan Blackford for her assistance with SCALAPACK, and Antoine Petitet for his very fast LU HPC tester and quite humorous commentary on different broken features of various MPI implementations.

References

1. I. Foster and C. Kesselman, *Globus: A Metacomputing Infrastructure Toolkit*, International Journal of Supercomputing applications, 11(2), PP 115-128.
2. K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, *A Resource Management Architecture for Metacomputing Systems*, in Proceedings of 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing, Workshop on Job Scheduling Strategies for Parallel Processing, 1998, Orlando, FL.
3. *The Globus Project*, http://www-fp.globus.org/overview/default.asp
4. *Globus Evaluation,* http://acts.nersc.gov/globus/evaluation.html
5. I. Foster, J. Geisler, W. Nickless, W. Smith, and S. Tuecke, *Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment*, in Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing, pp 562-571, 1996.
6. I. Foster, C. Kesselman, and S. Tuecke, *The Nexus Approach to Integrating Multithreading and Communication*, Journal of Parallel and Distributed Computing, 37, pp 70-82, 1996
7. I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill, *Remote I/O: Fast Access to Distant Storage*, in Proceeding of Workshop on I/O in Parallel and Distributed Systems, pp14-25, 1997
8. R. Thakur, W. Gropp, and E. Lusk, *An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces*, in Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, October 1996.
9. *Information Power Grid, http://www.nas.nasa.gov/IPG/index.html*
10. *Legion: A Worldwide Virtual Computer*, http://www.cs.virginia.edu/~legion
11. A. Grimshaw, A. ferrari, F. Knabe, and M. Humphry, *Legion: An Operating System for Wide-Area Computing*, Technical Report CS-99-12, University of Virginia, 1999
12. *Legion 1.6.3 System Administration Manual*, http://www.cs.virginia.edu/~legion/documentation.html
13. *Globus Quick Start Guide*, http://www-fp.globus.org/documentation/quick_start.html
14. M. Lewis and A. Grimshaw, *The Core Legion Object Model*, , Technical Report CS-95-35, University of Virginia, 1995
15. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke, *A Directory Service for Configuring High-Performance Distributed Computations*, in Proceedings of 6th IEEE Symposium on High-Performance Distributed Computing, pp 365-375, Portland, OR, 1997
16. P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski, *A Fault Detection Service for Wide Area Distributed Computations*, in Proceedings of the 7th IEEE Symposium on High Performance Distributing Computing, pp 268-278, 1998.
17. A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr., *Legion: The Next Logical Step Toward a Nationwide Virtual Computer*, Technical Report CS-94-21, University of Virginia, 1994
18. A. Ferrari, M. Lewis, C. Viles, A. Nguyen-Tuong, and A. Grimshaw, *Implementation of the Legion Library*, Technical Report CS-96-16, University of Virginia, 1996.
19. A. Grimshaw, A. NguyenpTuong, and W. Wulf, *Campus-Wide Computing: Early Results Using Legion at the University of Virginia*, Technical Report CS-95-19, University of Virginia, 1995.
20. W. Wulf, C. Wang, and D. Kienzle, *A New Model of Security for Distributed Systems*, Technical Report CS-95-34, University of Virginia, 1995.
21. A. Grimshaw, M. Lewis, A. Ferrari, and J. Karpovich, *Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems*, Technical Report CS-98-12, University of Virginia, 1998.
22. I. Foster and C. Kesselman, *The Globus Project: A Status Report*, in Proceedings of 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing, Heterogeneous Computing Workshop, 1998, Orlando, FL.
23. *About the Globus Security Infrastructure*, http://www-fp.globus.org/security/gssapi_ssleay.html
24. A. Ferrari, F. Knabe, M. Humphrey, S. Chapin, and A. Grimshaw, *A Flexible Security System for Metacomputing Environments*, Technical Report CS-98-36, University of Virginia, 1998.
25. S. Chapin, C. Wang, W. Wulf, F. Knabe, and A. Grimshaw, *A New Model of Security for Metasystems*, http://www.cs.virginia.edu/~legion/papers/LegionSecModel.ps
26. S. Chapin, D. Kataramatos, J. Karpovich, and A . Grimshaw, *Resource Management in Legion*, Technical Report CS-98-09, University of Virginia, 1998.
27. *Globus Installation Guide*

28. J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke, *GASS: A Data Movement and Access Service for Wide Area Computing Systems*, in Proceedings of 6th Workshop on I/O in Parallel and Distributed Systems, 1999.
29. *Legion Basic Userś Manual* , http://www.cs.virginia/~legion/documentation.html
30. *Legion 1.6 Reference Manual*, http://www.cs.virginia/~legion/documentation.html
31. A. Ferrari and A. Grimshaw, *Basic Fortran Support in Legion*, Technical Report CS-98-11, University of Virginia, 1998.
32. G. Lindahl, S. Chapin, N. Beekwilder, and A. Grimshaw, *Experiences with Legion on the Centurion Cluster*, Technical Report CS-98-27, University of Virginia, 1998.
33. A. Grimshaw, W. Wulf, J. French, A. Weaver, and P. Reynolds, Jr., *A Synopsis of the Legion Project*, Technical Report CS-94-20, University of Virginia, 1994.
34. A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey, *Legion: An Operating System for Wide-Area Computing*, Technical Report CS-99-12, University of Virginia, 1999.
35. S. Brunett, K. Czajkowski, S. Fitzgerald, I. Foster, A. Johnson, C. Kesselman, J. Leigh, and S. Tuecke, *Application Experiences with the Globus Toolkit*, in Proceeding of the 7th IEEE Symposium on High Performance and Distributed Computing, 1998.
36. I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, *A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation*, in Proceedings of International Workshop on Quality of Service, 1999.
37. Graham E. Fagg, Kevin S. London and Jack J. Dongarra, *MPI_Connect, Managing Heterogeneous MPI Application Interoperation and Process Control*, in Processing of the EuroPVM-MPI 98 meeting, Lecture Notes in Computer Science, Vol. 1497, pp.93-96, Springer Verlag, 1998.
38. Marty Humphrey, Frederick Knabe, Adam Ferrari, and Andrew Grimshaw. "Accountability and Control of Process Creation in Metasystems". Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS2000), San Diego, CA, February 2000.
39. Steve Chapin, John Karpovich, Andrew Grimshaw, "The Legion Resource Management System" Proc. 5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99) April 1999.
40. Edgar Gabriel, Michael Resch and Roland Ruehle 'Implementing MPI with Optimized Algorithms for Metacomputing' in 'Proceedings of the Third MPI Developer's and User's Conference', MPI Software Technology Press, Starkville Mississippi, 1999.

Appendix A

SCALAPACK Input parameter file for the LU solver (LU.dat)

```
'MPICH-G'
'LU.out'       output file name (if any)
25                  device out
5                   number of problems sizes
1000 2000 4000 5000 6000    values of M
1000 2000 4000 5000 6000    values of N
1                   number of NB's
40                  values of NB
1                   number of NRHS's
1                   values of NRHS
1                   Number of NBRHS's
1                   values of NBRHS
1                   number of process grids (ordered pairs of P & Q)
2 1 4               values of P
2 4 1               values of Q
1.0                 threshold
F                   (T or F) Test Cond. Est. and Iter. Ref. Routines
```

Appendix B

Input parameter data file for the HPC tester (HPC.dat)

```
HPC Version 1.0, Linpack benchmark input file
Innovative Computing Laboratory, University of Tennessee, 1999
HPC.out         output file name (if any)
6               device out (6=stdout,7=stderr,output file otherwise)
15              number of problems sizes
1000 1000 1500 1500 2000 2000 2500 3000 4000 5000 6000 7000 8000 9000 10000 values of N
1               number of NB's
40              values of NB
1               number of process grids (ordered pairs of P & Q)
2               values of P
2               values of Q
-16.0           threshold
1               number of panel fact. (PFACT)
2               values of PFACT (0=left looking, 1=crout, 2=right looking)
1               number of recursive stopping criterium (NBMIN >= 1)
4               values of NBMIN
1               number of panels in recursion (NDIV >= 2)
2               values of NDIV
1               number of recursive panel fact. (RFACT)
1               values of RFACT (0=left looking, 1=crout, 2=right looking)
1               number of broadcast (BCAST)
2               values of BCAST (0=I-rg, 1=I-2rg, 2=I-rg', 3=I-2rg', 4=S-rg)
1               number of lookahead depth (DEPTH)
0               values of DEPTH (>=0 - 0 = no lookahead)
4               memory alignment in double precision words (> 0)
```

Appendix C Globus MetaApplications and Projects

SF-Express

Very large distributed interactive simulation. This program holds the record for largest ModSA F run ever done. This was actually run across a number of DoD MSRC sites.

NEPH

Remote retrieval and processing of DMSP satellite data to determine cloud position and elevation. This is of direct consequence to DoD mission.

SpaceJunk

Remote retrieval and trajectory determination of objects in space and rockets being launched. Of direct consequence to DoD mission.

Cactus (Various groups)

Solves Einstein equations to simulation black hole collisions. Largest runs have spanned multiple large-scale supercomputers.

Overflow (NASA Ames: Djomehri)

Large CFD calculation that is used for aeronautic design.

PTOMO

Online tomographic reconstruction and display of instrumentation data.

Appendix D Legion MetaApplications and Projects

Aerospace
Overflow (Boeing: Kerlick)
Large CFD calculation that is used for aeronautic design.
Design Explorer (Boeing)
A heuristic parameter-space tool.
flapper (UVa - Lewin)
A study of the aerodynamics of flapping flight, using numerical models to simulate insect flight.

Biochemistry & molecular science
complib (UVa - Pearson/Grimshaw)
A comparison of protein DNA sequences, developed by William Pearson at the University of Virginia Biochemistry Department.
FASTA
Smith-Waterman
These two are DNA sequence comparison codes, used w/complib.
gnomad (Stanford - Altman/Williams)
A C code that compiles 3 input files to find the optimal configuration of a molecule.
feature (Stanford - Altman)
A C++ code that matches characteristics of proteins to determine probability that unknown protein will bind to DNA, what functions it has, etc.
CHARMM (Scripps - Brooks)
Chemistry at Harvard Molecular Mechanics (CHARMM). A program for macromolecular dynamics and mechanics, designed to investigate the structure and dynamics of large molecules. The software in this particular application was developed by Charles Brooks at the Scripps Research Institute.

Astronomy
Hydro code (Hawley/Holcomb)
An astronomical code used for simulating gas accretion disks.

Materials Science
DSMC - Direct Simulation Monte Carlo (UVa - Wadley/Beekwilder)
Developed by G.A. Bird and modified for the Directed Vapor Deposition research at the University of Virginia's Intelligent Processing of Materials Laboratory.
Large scale molecular dynamic (NAVO-LSU)

Information Retrieval
PIE: Personalized Information Environments (French & Viles)
A tool to create user-customized collections of information resources based on distributed search over restricted search spaces; virtual information repositories; and a novel system architecture. It also provides for user anonymity and secure access to resources.

Climate, Weather, Ocean
BT-MED (NAVO - Piechezk)
Code provided by Northrop-Grumman(NAVO)
MM5 (NCAR)
A mesoscale weather modeling code used for both research into weather prediction programs and for operational predictions for organizations such as the United States Air Force.
NLOM-COAMPS - (NAVO - Bettencourt)
A coupled application project: NLOM is Navy Layered Ocean Model and COAMPS is Coupled Ocean/Atmosphere Mesoscale Prediction System.

Neuroscience

Biological scale simulations of a mammalian neural network (UVa - Levy)