Capacity and Capability Computing using Legion

Anand Natrajan, Marty Humphrey, Andrew Grimshaw
Department of Computer Science, University of Virginia, Charlottesville, VA 22904
{anand, humphrey, grimshaw}@cs.virginia.edu

Abstract. Computational Scientists often cannot easily access the large amounts of resources their applications require. *Legion* is a collection of software services that facilitate the secure and easy use of local and non-local resources by providing the illusion of a single virtual machine from heterogeneous, geographically-distributed resources. This paper describes the newest additions to Legion that enable high-performance (capacity) computing as well as secure, fault-tolerant and collaborative (capability) computing.

1 Introduction

As available computing power increases because of faster commodity processors and faster networking, computational scientists are attempting to solve problems that were considered infeasible until recently. However, merely connecting large machines with high-speed networks is not enough; an easy-to-use and unified software environment in which to develop, test and conduct software experiments is absent. For example, users often are forced to remember multiple passwords, copy files to and from machines, determine where necessary compilers and libraries are on each machine, and choose which machines to use at particular times.

A *metasystem* is an environment in which users, such as scientists, can access resources in a transparent and secure manner. In a metasystem, users are not limited by geography, by non-possession of accounts, by limits of resources at one site or another and so on. In short, as long as a resource provider is willing to permit a user to use the resource, there is no barrier between the user and the resource.

Legion is an architecture for a metasystem [1]. Just as an operating system provides an abstraction of a machine, Legion provides an abstraction of the metasystem. This abstraction supports the current performance demands of scientific applications. A number of scientific applications already run using Legion as the underlying infrastructure. In the future, scientists will demand support for new methods of collaboration. Legion supports these expected demands as well.

We define *capacity computing* loosely as the ability to conduct larger computational experiments either by expending more resources on a single problem or on multiple, independent problems. We define *capability computing* to be new mechanisms with which to conduct computational science experiments. This paper describes, from the viewpoint of a computational scientist, Legion's unique support for

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

high-performance capacity and capability computing and describes how computational scientists in a variety of disciplines are using Legion today.

2 Legion

The Legion project is an architecture for designing and building system services that present users the illusion of a single virtual machine [2]. This virtual machine provides secure shared objects and shared name spaces. Whereas a conventional operating system provides an abstraction of a single computer, Legion aggregates a large number of diverse computers running different operating systems into a single abstraction. As part of this abstraction, Legion provides mechanisms to couple diverse applications and diverse resources, vastly simplifying the task of writing applications in heterogeneous distributed systems.

Each system and application component in Legion is an object. The object-based architecture enables modularity, data and fault encapsulation and replaceability — the ability to change implementations of any component. Legion provides persistent storage, process management, inter-process communication, security and resource management services, long regarded as the basic services any operating system must provide. Legion provides these services in an integrated environment, not as disjoint mechanisms such as Globus does [3]. Of particular importance is the integration of security into Legion from the design through implementation. Legion supports PVM [4], MPI [5], C, Fortran (with an object-based parallel dialect), a parallel C++ [6], Java and the CORBA IDL [7]. Also, Legion addresses critical issues such as flexibility and extensibility, site autonomy, binary management and limited forms of fault detection/ recovery. From inception Legion was designed to manage millions of hosts and billions of objects — a capability lacking in other object-based distributed systems [8].

3 Capacity Computing with Legion

Legion can benefit scientific applications by delivering large amounts of resources such as computing power, storage space and memory. Moreover, Legion provides a rich set of tools that make the access and use of these resources simple and straightforward. In particular, there are tools for running programs written using MPI and PVM as well as programs that are parameter-space studies or sequential codes. In §3.1-§3.4, we present some of Legion's tools for running applications. In §3.5, we discuss scheduling in Legion briefly.

3.1 Legacy Applications

Legacy applications are those whose source code does not consist of any calls to Legion routines and does not utilise Legion objects and tools. Moreover, the source code of the application may not be modified to target it to Legion, either because it is unavailable or because its authors are unavailable or unwilling to make the necessary

changes. In all such cases, Legion neither mandates re-targetting the application nor denies access to metasystem resources.

A Legion user may run a legacy application on the distributed resources of a metasystem by undertaking two steps (tool names are in parentheses):

1. Register the executable as a runnable class (`legion_register_program`)
2. Run the class (`legion_run`)

The first step results in the creation of a *runnable class*, analogous to an executable in Unix or Windows. Registering an executable is an infrequent step, required only when the runnable class does not exist in Legion or when the executable available to the user changes. A user is likely to execute the second step repeatedly in order to initiate, monitor and complete repeated runs of the application. The executable registered with this class is called an *implementation*. Multiple executables, typically of different architectures, may be registered with the same class, for example:

```
legion_register_program myClass /bin/whoami solaris
legion_register_program myClass /bin/ls sgi
```

The first command creates a class `myClass`, which Legion tools can recognise as a runnable class. The second parameter to the command is the Unix (or Windows) path of an executable to be registered as an implementation for `myClass`. The third argument indicates that the executable is a Solaris binary. When the second command is executed, Legion recognises that `myClass` exists. It adds the binary `/bin/ls` as an SGI implementation for the same class. Subsequently, if a user runs `myClass` on a Solaris machine, the executable corresponding to `/bin/whoami` will be executed on that machine, whereas if the user runs `myClass` on an SGI machine, the executable corresponding to `/bin/ls` will be executed on that machine.

This example is trivial in the sense that `/bin/whoami` and `/bin/ls` are not high-performance applications. Moreover, running `myClass` on different architectures is likely to give very different results. However, the example illustrates that (a) registering legacy applications in Legion is simple and (b) no semantic requirement is imposed on the executables registered for different architectures.

Once a runnable class has been created in Legion, a user can run the class by issuing a `legion_run` command. The simplest form of the command is:

```
legion_run myClass
```

Here, the user implies that Legion can run an instance of the class on any resource present in Legion provided (a) `myClass` has implementations for the machine on which the instance eventually runs (i.e., Solaris or SGI implementations), (b) the user is permitted to run on the machine and (c) the machine accepts the instance for running. A more sophisticated run is:

```
legion_run -v -IN file1 -OUT file2 myClass convert
```

Here, the user indicates that she will observe the run in verbose mode (`-v`), will provide one Unix or Windows input file (`-IN file1`) and will receive one Unix or Windows output file (`-OUT file2`) when running an instance of `myClass` with the argument `convert`. Legion ensures that the input and output files are copied to and from the machine on which the instance runs. In this form as well, the user has indicated that she prefers Legion to select the machine on which the instance runs. While this transparency in scheduling is used often, some users happen to be aware of

the machine on which they would like to run. Therefore, Legion permits directed scheduling, wherein the user specifies the machine on which she wants to run:

```
legion_run -h /hosts/xyz -IN file1 -OUT file2 myClass convert
```

The details of how individual runs can be configured to suit a user's requirements are beyond the scope of this paper. It suffices to say that in keeping with the Legion philosophy of providing mechanisms on which policies can be constructed, there exist many different strategies for executing a legacy application on distributed resources. These different strategies can be applied by choosing from a large number of options available in `legion_run`. The options are part of the standard documentation and man pages available at each Legion installation [9].

3.2 MPI Applications

Many high-performance parallel applications are written using the Message Passing Interface (MPI) library [5]. An MPI library provides routines that enable communication among various processes of a parallel application. MPI is a standard, i.e., it defines the interface of the routines. Different vendors of MPI may implement a routine differently provided they adhere to the standard interface. Legion's support for MPI is three-fold: Legion MPI, native MPI and mixed MPI.

Legion MPI. Legion can be viewed as another MPI vendor because it provides implementations to standard MPI routines. If a user desires that an application using MPI routines should run on a metasystem, he has to undertake three simple steps:

1. Re-link the object code of the application with Legion libraries (`legion_link`)
2. Register the executable as an MPI runnable class (`legion_mpi_register`)
3. Run the class (`legion_mpi_run`)

The first step ensures that Legion's implementation of MPI routines are used when running the application. Note that it is not necessary to change the source code of the application. The subsequent steps are similar to those for legacy applications. The options and operations of the actual commands are similar to those for registering and running legacy applications.

Native MPI. Some MPI applications are intolerant of high latencies for inter-process communications. Running such applications on distributed resources may degrade the performance of the application. Such applications are better supported by running them on proximal resources to reduce communications latency. Moreover, many MPI implementations are tuned finely to exploit the architecture of underlying resources. Finally, the users of many MPI applications may be unwilling or unable to re-link the application with Legion libraries. Therefore, Legion supports running MPI applications in "native" mode, i.e., using other implementations of MPI, such as MPICH [10]. Native MPI support is similar to support for Legion MPI as well as legacy applications. The steps a user has to undertake are:

1. Register the executable as a runnable class (`legion_native_mpi_register`)
2. Run the class (`legion_native_mpi_run`)

The benefits to the user are that no recompiling or re-linking is necessary to access remote resources in a transparent manner.

Mixed MPI. Mixed MPI support is a blend of Legion MPI and native MPI. In Legion's mixed MPI support, an application is executed in "native" mode, but the application can access Legion's objects, such as files. The steps required are:

1. Modify source code to initialize Legion library
2. Re-link the object code of the application with Legion libraries (`legion_link`)
3. Register the executable as a runnable class (`legion_native_mpi_register`)
4. Run the class (`legion_native_mpi_run -legion`)

The user has to modify the source code to initialize Legion with one call from within the application. Registering and running the class is similar to native MPI with the addition of one option. Applications written to take advantage of mixed MPI support can benefit in two ways: (a) since runs are executed in native mode, performance for latency-intolerant applications does not suffer, and (b) runs can access Legion objects and thus take advantage of the metasystem.

3.3 Mentat and Basic Fortran Support (BFS)

High-performance applications can be supported in Legion if they are written in Mentat or if they use the Basic Fortran Support. Mentat is a language similar to C++ with a few additional keywords [6]. In Mentat, users may specify classes to be stateless or persistent. The Mentat compiler identifies data dependencies within a program and constructs a dataflow graph to execute the program. Mentat provides a platform for users to write high-performance applications using a compiler constructed to mask the tedium of writing parallel programs. Legion's support for Fortran programs is called BFS [11]. If users desire to write metasystem applications in Fortran, then Legion requires that metasystem directives be embedded within Fortran comments. Currently, BFS support targets Mentat, but may not in future releases.

3.4 Parameter-Space Studies

Many metasystem applications are parameter-space (p-space) studies. In a p-space study, a single program is called repeatedly with different sets of parameters. Multiple instances of the program may run concurrently with different sets of parameters. These instances are completely independent of one another. Therefore, they can be scheduled easily across geographically-distributed resources.

With Legion's support, users may run their p-space studies orders of magnitude faster than sequential. First, the application must be registered (see §3.1-§3.2). Next, the user must indicate which files must be mapped to the files required by an instance. Finally, the application must be run with `legion_run_multi`. Legion runs each instance of the application by mapping the proper files for the instance and copying output files appropriately. `legion_run_multi` takes a number of options in order to tailor the running of a p-space application for a user. This tool ensures that input files and output files are arranged such that the user can identify corresponding sets easily.

3.5 Scheduling

In a metasystem, scheduling is the process of initiating runs on the best possible resources. The general scheduling problem is NP-complete [12]. In addition, the parameters involved in making an optimal schedule are numerous and mutually dependent. Constructing a schedule may involve making decisions not limited to: (a) the machine architectures for which a class has implementations, (b) specific properties of a machine desired by the class (e.g., is it a queuing system? can it run MPI jobs natively?), (c) communication bandwidth *versus* performance penalty, (d) current load and storage space on the machine, (e) permissions for this user to run an instance of this class on that machine, (f) allocation remaining for the user on that machine and (g) charges imposed by resource providers for running on their machine.

Legion provides mechanisms to construct schedulers. Different schedulers may employ different algorithms to construct schedules from the list of available resources. Also, Legion permits users to specify resources directly for a run, the rationale being that until good heuristics are developed to address all issues in scheduling, users are likely to be the best schedulers of their own jobs.

The general scheduling architecture in Legion is based on negotiation between resource providers and consumers [13]. The negotiation process preserves autonomy of resource providers while satisfying the demands of the consumers. When a user starts a run, Legion encapsulates the demands of the user in the run request. The scheduler uses this request to construct one or more schedules for this run. Next, it queries the resource objects in turn to determine if they will accept the run. The resource objects may exercise the autonomy of the resource providers in accepting or denying the run. If they accept, the runs are initiated on the chosen resources.

4 Capability Computing with Legion

A well-designed metasystem should not only satisfy current demands of users but also anticipate and satisfy future demands. Currently, many applications require high performance. However, in the near future, metasystems such as Legion will be able to deliver high performance to applications routinely by providing access to distributed resources. We believe that at that point, users will look beyond high performance as the defining feature of a metasystem. At that point, users' demands may include heterogeneity, security, fault-tolerance and collaboration.

Heterogeneity is a fundamental design principle in Legion [14]. Typically, a running metasystem that uses Legion incorporates diverse resources — machines of different architectures running different operating systems consisting of different configurations and managed by different organisations. As in §3, users may register implementations of different architectures for their runnable classes. For parallel applications, different instances started by a single run may run on heterogeneous machines and communicate with one another as if they ran on homogeneous machines.

Security was designed into Legion from the start [15]. Every Legion object, whether it be a resource, a user, a file, a runnable class or a running instance, has a security mechanism associated with it. The mechanisms provided by Legion are

general enough to accommodate different kinds of security policies within a single metasystem. Typically, the security provided is in the form of access control lists. An access control list indicates which objects can call which methods of an object. This fine-grained control mechanism enables users and metasystem administrators to set sophisticated policies for different objects. The authentication mechanism currently employed by Legion is a public key infrastructure based on key pairs. The keys are used to encrypt and decrypt messages securely as well as for signing certificates.

Fault-tolerance can be implemented in a number of ways in Legion [16]. Basic Legion objects are fault-tolerant because they can be deactivated at any time. When a Legion object is deactivated, it saves its state to persistent storage and frees memory and process state. Subsequently, it may be reactivated from its persistent state either on the same or a different machine. If it is reactivated on a different machine, Legion transfers its state to the new machine whenever possible. In addition, some objects can be replicated for performance or availability. Legion's MPI implementation provides mechanisms for checkpointing, stopping and restarting individual instances. Finally, Legion provides tools for retrieving intermediate files generated by legacy applications. Users can restart their instances using these intermediate files.

Legion enables new paradigms for collaboration between researchers conducting experiments that require using metasystem resources. We believe that collaboration is an important goal for a metasystem. We expect that researchers should not be limited by geographical distance between one another as well as the resources they desire to use. Accordingly, the ability to share objects via their permissions (access control lists) has always been a key design feature in Legion. In §4.1 and §4.2, we outline some of the methods by which users of a metasystem can collaborate.

4.1 Context Space

Legion provides a shared, virtual space to metasystem users. The shared, virtual space can be viewed as a truly distributed, global file system. This file system is organised in a manner similar to a Unix file system. In order to distinguish the global file system from the file systems present on individual machines, we call the global file system a *context space*. Directories in context space are called *contexts*. A context called "*/*" typically denotes the root of the context space. A context is an object that contains other objects — contexts, machines, users, classes, files, etc. All users of a metasystem, no matter where located physically, have the same view of the context space. The analogue of this model in traditional operating systems is an NFS-mounted disk that is visible to all machines that share the mount, or a Samba-mounted Unix directory that is visible from a Windows machine.

The scope of Legion's context space is much vaster than that of any of its predecessors. Distributed file systems are not novel. Legion's implementation has predecessors in Network File System (NFS) [17], the Andrew File System (AFS) [18] and Extensible File System (ELFS) [19]. However, context space is truly distributed and global; individual components may be physically located on machines that do not have anything in common except that they are part of the same metasystem.

Users may freely transfer files from their local file systems to context space. For example, one of the options to a tool called `legion_cp` permits users to copy a text

file from their file system to context space. Likewise, registering a program effectively transfers an executable from a local file system to context space. A growing number of tools available in Legion permit users to interface with context space in novel ways. For example, a tool called `legion_export_dir` lets a user mirror an entire directory in his local file system into Legion. Likewise, a Windows tool lets users browse context space. When these two tools are used in conjunction, a user on one Windows machine may be able to view the contents of his collaborator's directories on another Windows machine across the globe. Naturally, the permissions on the exported directory and its components have to be set to permit the collaborator (and perhaps only the collaborator) to view them. However, setting the permissions is a matter of manipulating the access control lists of the objects. Legion provides tools for manipulating the access control lists of objects.

Tools for traversing context space include a suite of Unix-like command-line tools, a point-and-click Web browser interface, an FTP tool, a Samba interface for Windows, an HTTP interface, and a Legion implementation of NFS for accessing context space with standard Unix tools such as `ls` and `cat` as well as with standard system calls like `open`, `read` and `write` [20]. Using these tools, metasystem users can collaborate by sharing and exchanging data in a manner familiar to them. Moreover, because of the possibility of setting fine-grained access controls, collaborators can also select the level of collaboration.

4.2 Sharing Runs

Legion's object model is flexible enough to permit novel means of collaboration among researchers, for example, sharing runs. In Legion, running instances of a class are first-class objects themselves. Therefore, as with any object in Legion, access control lists can be set for them to control permissions in interesting ways.

Suppose two researchers situated across a country wish to collaborate. The nature of their collaboration requires one of them to initiate a run which both observe. Currently, such a collaboration would be impossible unless both researchers were able to share an account on some machine. In Legion, neither researcher would need an account on the machine on which the instance runs. Instead, both could access the same object using Legion tools from their own machines.

Suppose a researcher constructs an application that is likely to be used widely by others in the same field. The researcher could register her executable as a runnable class in Legion and set the permissions to allow anyone, a group of users or an *a priori* known set of users to run instances of the class. Currently, the researcher would have to send or sell her executable to her fellow researchers. In the Legion model, she could control who runs her class when, where and how many times without physically transporting her executable to the other researchers' machines.

Suppose two mutually distrustful parties wish to collaborate on an experiment with one providing the executable and the other the data. Currently, such a collaboration is impossible because either the executable or the data must be transported to the other collaborator. However, in Legion, such a collaboration is legitimate and possible. The collaborator with the executable would register the executable as a class in Legion and start an instance. Then he would set the

permissions on the instance allowing only the other collaborator to perform data transfers but retaining permission to terminate the experiment. The second collaborator, after verifying that the permissions are indeed as outlined above, could commence transferring data files. The application in question would have to be written in such a manner that it can wait until the data files become present. With that minor change in place, Legion can enable these mutually distrustful parties to collaborate.

Other means of collaboration will become evident as metasystems are used more widely and routinely. We expect the Legion model to be flexible enough to accommodate these collaboration efforts as they arise.

5 Conclusions and Current Status

The success of a metasystem depends on how easily and securely it permits users to perform their computations by collaborating and accessing available resources. A key component of a metasystem is software that presents users with abstractions of resources. Legion provides those abstractions *via* uniform, easy-to-use interfaces. These interfaces, ranging from tool-level to programming-level, greatly reduce the difficulties of computing in distributed, heterogeneous environments. The mechanisms underlying the interfaces enable users to perform cross-machine, cross-architecture and cross-organisation computation. By enabling such computations on a large scale, Legion supports capacity computing. Legion's flexible and extensible object model supports capability computing by permitting novel methods of computation.

Legion consists of 350,000 lines of code and has been ported to Windows NT as well as a large number of Unix variants, including Linux (Intel, Alpha), Unicos (T90, T3E), AIX (SP-2, SP-3), HPUX, FreeBSD, IRIX (Origin 2000) and Solaris (Enterprise 10000). Legion has been integrated with a large number of queuing systems, such as PBS, LSF, Codine, LoadLeveler and NQS. It has been deployed on machines belonging to NSF-PACI, NASA IPG and the DoD MSRCs. Currently, Legion is running at over 300 hosts across the United States and Europe. Researchers using Legion currently are from a number of disciplines, including:

- Biochemistry (e.g., complib, a protein and DNA sequence comparison)
- Molecular Biology (e.g., CHARMM, a p-space study of 3D structures)
- Materials Science (e.g., DSMC, a Monte Carlo particle-in-cell study)
- Aerospace (e.g., flapper, a p-space study of a vehicle with flapping wings)
- Information Retrieval (e.g., PIE, a personalised search environment)
- Climate Modelling (e.g., BT-MED, a 2D barotropic ocean model)
- Astronomy (e.g., Hydro, a study of a rotating gas disk around a black hole)
- Neuroscience (e.g., a biological-scale simulation of a mammalian neural net)
- Computer Graphics (e.g., a parallel rendering of independent movie frames)

We expect users to become more accustomed to using distributed resources, often in ways not anticipated today. Legion's architecture holds the promise of satisfying the computational demands of the present as well as the future.

6 References

1. Grimshaw, A. S., Ferrari, A. J., Lindahl, G., Holcomb, K., *Metasystems*, Communications of the ACM, Vol. 41, No. 11, November 1998.
2. Grimshaw, A. S., Wulf, W. A., *The Legion Vision of a Worldwide Virtual Computer*, Communications of the ACM, Vol. 40, No. 1, January 1997.
3. Foster, I., Kesselman, C., *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.
4. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1998.
5. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J., *MPI: The Complete Reference*, MIT Press, 1998.
6. Grimshaw, A. S., Ferrari, A. J., West, E., *Mentat*, Parallel Programming Using C++, MIT Press, 1996.
7. Seigel, J., *CORBA Fundamentals and Programming*, Wiley, ISBN: 0471-12148-7, 1996.
8. Grimshaw, A. S., Ferrari, A. J., Knabe, F., Humphrey, M. A., *Wide-Area Computing: Resource Sharing on a Large Scale*, IEEE Computer, Vol. 32, No. 5, May 1999.
9. —, *The Legion Manuals (v1.7)*, University of Virginia, October 2000.
10. Gropp, W., Lusk, E., Doss, N., Skjellum, A., *A High-Performance, Portable Implementation of the Message Passing Interface Standard*, Parallel Computing, Vol. 22, No. 6, September 1996.
11. Ferrari, A. J., Grimshaw, A. S., *Basic Fortran Support in Legion*, University of Virginia Technical Report CS-98-11, March 1998.
12. Weissman, J., *Scheduling Parallel Computations in a Heterogeneous Environment*, Ph.D. Dissertation, University of Virginia, August 1995.
13. Chapin, S. J., Katramatos, D., Karpovich, J. F., Grimshaw, A. S., *Resource Management in Legion*, University of Virginia Technical Report CS-98-09, February 1998.
14. Grimshaw, A. S., Lewis, M. J., Ferrari, A. J., Karpovich, J. F., *Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems*, University of Virginia Technical Report CS-98-12, June 1998.
15. Ferrari, A. J., Knabe, F., Humphrey, M. A., Chapin, S. J., Grimshaw, A. S., *A Flexible Security System for Metacomputing Environments*, High Performance Computing and Networking Europe, April 1999.
16. Nguyen-Tuong, A., *Integrating Fault-tolerance Techniques in Grid Applications*, Ph.D. Dissertation, University of Virginia, August 2000.
17. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B., *Design and Implementation of the SUN Network File System*, Proceedings of USENIX Conference, 1985.
18. Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988.
19. Karpovich, J. F., Grimshaw, A. S., French, J. C., *Extensible File Systems (ELFS): An Object-Oriented Approach to High Performance File I/O*, 9th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), October 1994.
20. White, B. S., Grimshaw, A. S., Nguyen-Tuong, A., *Grid-Based File Access: The Legion I/O Model*, High Performance Distributed Computing 9, August 2000.