Runtime Support for Automatic Wide Area Implementation Management in Legion

Michael J. Lewis
Binghamton University (SUNY)
Binghamton, NY 13902

Andrew S. Grimshaw
University of Virginia
Charlottesville, VA 22903

Abstract

Computational grids have the potential to give programmers access to a much larger set and wider range of underlying computing resources than currently possible. To be useful in this capacity, however, grids must address the problems of heterogeneity, multiple disjoint administrative domains, and scale. This paper describes the Legion implementation model, which prescribes how active objects are created and managed in the face of these problems. The Legion wide area metasystem maintains object executables in architecture-specific binary format, and automatically downloads and executes the appropriate binary for the platform on which the object is to run. The implementation model relieves the programmer from requiring an individual account on all the machines or domains on which her objects may run, and from explicitly installing programs throughout the metasystem. Instead, programmers have a single account, register executables once, with Legion, and can then potentially utilize all the machines in the system.

1. Introduction

The recent proliferation of computing resources, along with their interconnection via high-speed low-latency computer networks, has made it possible to federate the resources into computational grids [3]. Among the promises of grid technology is the ability for programmers to access and utilize far more computing power than ever before. For some users and applications, this benefit will be gained simply by being able to choose from a larger set of more diverse computing resources. For others, harnessing the power of computational grids requires that the grids be able to support parallel processing systems efficiently; that is, computational grids must support, or act themselves as, run-time systems for parallel and distributed object programming.

Legion is an approach to computational grid technology that defines and implements a uniform architecture and object model [9]. Legion is an object-oriented system comprising independent, logically-address-space-disjoint objects, which communicate with one another via remote method invocation. Legion implements a macro-data flow method execution model; inter-object function calls are non-blocking and may be accepted in any order by the called object. Each function has a signature that describes its parameters and return values (if any); the complete set of signatures for an object describes that object’s interface, which is determined by its class.

From a programmer’s perspective, one of Legion’s roles is to act as a runtime system that is intended to be the target of multiple different parallel and distributed processing models and languages. For example, Legion currently supports object-oriented parallel languages such as Mentat [5], message passing systems such as PVM [4] and MPI [6], and object-based programming using CORBA-style stub generation [11]. Alternatively, a programmer can simply submit a sequential program, along with a set of execution constraints (e.g. the architecture or specific host on which it must run), and Legion will schedule it on an appropriate host and return the results back to the user. A basic core requirement for supporting any of these models is to be able to create a new active entity in the system. This is true whether that entity, from the programmer’s view, is an object that responds to method invocations, a process running on an underlying operating system, or a thread that runs within the address space of an existing process.

Legion’s approach is to support each of the different programming models using objects. That is, an object is the abstraction of an active entity in Legion, regardless of the particular abstraction that is presented to the programmer. The job of the compiler and Legion-targetting run-time library is to implement, using Legion objects, the program-
moming abstraction that they support. In the case of Mentat and CORBA, this is quite straightforward. Mentat and CORBA objects map directly to run-time Legion objects, and the methods they accept are exactly those methods that are specified by the programmer in her code. For MPI and PVM, instances of the task abstraction are implemented by special Legion objects which are built to support methods such as send() and receive() [7]. The point is that Legion objects provide the basis for Legion to be the runtime system for the various programming models, and can be used to support others.

This paper presents the Legion implementation model, which describes the mechanism by which Legion supports the object as the abstraction of an active entity. The model is specifically designed to deal with the challenges of Legion's environment. In particular, the model deals with problems due to heterogeneity, disjoint administrative domains (with separate filesystems, user accounts, etc.), and scale. In subsequent sections we identify these challenges more precisely, describe the core Legion implementation model and how it is intended to meet these challenges, and discuss how the model has been extended to support component-based object development.

2. Challenges

At the lowest level, Legion objects are typically represented by processes that run directly on top of host operating systems. In computational grids, which exhibit a large degree of heterogeneity, the mechanism for creating a new process can vary widely for the different execution platforms that exist in the system. The complexity due to heterogeneity goes beyond simply a difference in machine architecture; the mechanisms for creating and monitoring processes differ with the underlying operating systems, and it is not uncommon for sites to require that access to computing resources be made only through a queuing system. From a user’s or programmer’s perspective, objects should simply run on the computational grid, and the necessarily different mechanisms for creating, managing, and moving them, due to heterogeneity, should be hidden by the system.

Also hidden from users should be the fact that a computational grid can span multiple administrative domains. These domains will necessarily require operating system specific user names and passwords, and will certainly not share file space with all other domains that implement part of the grid. Being able to run a program on some particular system requires that the executable for that program eventually be downloaded to a file that can be accessed by the local OS process creation mechanism. Users should not have to do this explicitly for all domains on which their objects might run, and should not have to manage multiple account names and passwords.

Finally, the mechanism for creating active entities should be designed to scale to the size of the environment for which it is intended to be used. Computational grids will potentially contain millions of hosts and trillions of simultaneously running objects/programs.

3. The Legion Implementation Model

The design of the Legion implementation model was driven by challenges of the computational grid environment, as described above, and by the desire to separate concerns and functionality into appropriate abstractions. Five different object types play roles in the Legion implementation model: class objects, host objects, scheduling agents, implementation objects, and implementation caches. An overview of the mechanism is presented below, followed by a more detailed discussion of the implementation model.

3.1 Overview

Implementation objects abstract binary executables in Legion. An implementation object responds to member function calls, like all other Legion objects, and allows other system objects to access the executable code used to create processes. Class objects maintain the names of several different implementation objects, one per architecture type supported by the class. The set of implementation objects typically (though not necessarily) results from the same source code being compiled for different architectures. In response to a request to create a new instance, or to move an instance to a new node, the class uses a scheduling agent to select the host on which the instance should run, and notifies the host object that represents that particular machine, giving it the name of an implementation object which is appropriate for the host’s architecture. Next, the host object uses an implementation cache to retrieve the executable code from the implementation object, and uses that executable to create the process that should represent the instance. Using an implementation cache allows the host to benefit from other “nearby” hosts downloading the same executable. The roles of, and relationships among, these object types are motivated and developed in more detail below.

3.2 Class Objects and Scheduling Agents

As mentioned earlier, each object in Legion has a type, which is defined by that object's class, and which determines the functions that the object responds to (its interface) and how it carries them out (its implementation).
A class is itself a Legion object, and therefore contains an active thread of control. A class serves not only as the definer of some particular type, but also as a manager for all objects of that type. Thus, the class performs some of the services necessary for maintaining its instances in the system. In particular, a class object is responsible for creating new instances of the type it defines, in response to a call to createInstance() (which creates a single instance of the class) or createVectorInstance() (which creates multiple instances). That is, class objects provide the run-time interface to programs that wish to create other Legion objects.

In order to create instances, a class object maintains a list of the implementation types that it supports. An implementation type describes both an execution platform and an executable that can run on that platform. SGI, Solaris, and WinNTx86 are examples of currently supported implementation types. So a Sparc10 running Legion on top of SunOS 5.7 is said to be of implementation type “Sparc”, as are the executables that can run on Solaris machines. The implementation management mechanism must ensure that the right kind of executable gets paired up with the selected execution platform so that an object can be created or activated. This requirement acts as a constraint on the object placement (i.e. scheduling) algorithm, which is implemented within a Legion scheduling agent. To satisfy this constraint, the scheduling agent can retrieve the implementation type from host objects (which abstract execution platforms), implementation objects (which abstract executable code), and class objects (which determine which platforms are supported by the object type). The scheduling agent can therefore consider this constraint among others when making object placement decisions.

### 3.3 Implementation Objects

For each of its supported implementation types, the class object maintains the Legion name of an implementation object that is directly used to create an instance on machines of that implementation type. Implementation objects encapsulate and maintain the code for Legion object executables. The fact that the implementation object is a Legion object means that it has a globally accessible name, can be represented by an active thread of control, and its contents can be accessed through member function calls by any other object in the system.

The executable itself is treated much like a Unix file (i.e. as an array of bytes), so an implementation object interface is similar to a Unix file interface: it contains functions including read(), write(), and sizeOf(). Implementation objects are write-once, read-many objects—no updates are permitted after the executable is initially stored in its containing implementation object. Therefore, there is no danger of replicated executables becoming inconsistent. When an executable does need to be replaced by a newer version, the new version of the executable will have a brand new implementation object in which to reside.

Implementation objects typically contain executable code for a single platform, but may in general contain any information necessary to instantiate an object on a particular host. For example, implementations might contain Java byte code, Perl scripts, or high-level source code that requires compilation by a host. Like all other Legion objects, implementation objects describe themselves by maintaining a set of attributes. In their attributes, implementation objects specify their execution requirements and characteristics which may then be exploited during the scheduling process. For example, an implementation object may record, in addition to the implementation type, a minimum set of target machine requirements, performance characteristics of the code, etc.

Implementation objects allow classes a large degree of flexibility in customizing the behavior of individual instances. For example, a class might maintain implementations with different time/space trade-offs and select between them depending on the currently available resources. To provide users with the ability to select their cost/performance trade-offs, a class might maintain both a slower, low-cost implementation and a faster, higher-cost implementation. This is similar to abstract and concrete types in Emerald [1].

### 3.4 Host Objects and Implementation Caches

Host objects abstract Legion computing resources. A host object is in charge of creating processes to represent Legion objects, and typically does so at the request of class objects, which pass the host the Legion name of an implementation object to download and run. Different host objects can exist for the different execution environments and process creation mechanisms. The current Legion implementation contains host objects for Unix machines, Windows NT machines, and machines that exist behind several different queuing systems.

Implementation caches, which are also Legion objects, help host objects avoid storage and communication costs by storing implementations for later reuse. If multiple host objects share access to some common storage device (e.g. an NFS-mounted file system), they may share a single cache to further reduce copying and storage costs. The interface to the implementation cache object is a single method that returns the path of a local file containing a given implementation object’s data (i.e. an executable). Host objects, rather than downloading implementations themselves, invoke getImplementation() on their local implementation cache object. The cache object either finds
that it already has a cached copy of the implementation or it downloads and caches a new copy. In either case, the cache object returns the executable’s path within the local filesystem to the host. In terms of performance, using a cached binary results in object activation being only slightly more expensive than running a program from a local file system.

Implementations can be pre-cached by applications by having them call the getImplementation() function directly before placing the application’s objects. This should be done in concert with the scheduling agent, so that the implementation is pre-cached on the sites where it will run.

The implementation model makes the invalidation of cached binaries a trivial problem. Since class objects specify the name of the implementation to use on each activation request, a class need only change its list of implementations to replace the old executable with the new one. The new version will be specified with future activation requests, and the old implementation will simply no longer be used and will time-out and be discarded from caches.

4. Illustrative Example

This section uses a simple illustrative Master / Slave style parallel programming example to describe in more detail the process by which programmers create objects and register and run them within Legion. It also describes the details of the implementation model and what happens “behind the scenes” in order to download executables to appropriate sites, and to execute the implementations to create objects.

4.1 Building Legion Programs

The first step for a Legion programmer is to write and compile code that is ready to run within Legion. In particular, this involves using a Legion-aware compiler, and linking the Legion run-time library into an executable program. This process is completed for each architecture (implementation type) on which the object should run. As described earlier, the original source code written by the programmer can be MPI, straight Legion code (i.e. code that makes calls on the Legion run-time library directly), Mentat programming language, or others. For this example, we’ll assume that the code is written in Mentat, and compiled for Solaris, Intel, and SGI architectures, as shown in Figure 1.

The Legion-aware Mentat compiler is a source-to-source compiler that uses a C++ back-end compiler such as g++, to create native executables. The executables are stored in the programmer’s local file system, one per architecture per object type. In this case, there would be six executables, three for the Master object type, and three for the Slave object type.

![FIGURE 1. A Legion-aware compiler builds executables that are ready to run in Legion.](image)

4.2 Registering Executables with Legion

Once the executables have been created, the programmer needs to create two class objects, one to represent the Master object type, and one for the Slave. This is typically done by invoking the “legion_create_class” command line tool, which can take the name of a Legion metaclass as a parameter. The command line program will invoke the createInstance() member function of the named metaclass to create a new class object. The class may be created anywhere within Legion (or a host could be selected by the programmer when invoking the program).

The executables then need to be wrapped in Legion implementation objects to be made available to the Master and Slave class objects, and to host objects and implementation caches. This process is typically accomplished by invoking a Legion command line program called “legion_create_implementation.” This program takes the name of the local executable, the implementation type, and the name of the class object as parameters, and operates in two phases. In phase I, it creates an instance of the Legion class called ImplementationClass. Again, this new implementation object may be located on any host in Legion (ImplementationClass and its scheduling agent decide where to place it). The command line program then uploads the bytes of the executable from the local executable file into the new object and sets the implementation type. The process, depicted in Figure 2, is completed separately for all six executables and typically is semi-automated by including invocations to the tool in the application build process (i.e. the makefile), so that once the executables are compiled, they are also registered with Legion.

Phase II of the legion_create_implementation program associates the new implementation objects with the appropriate Master and Slave class objects. The executables will be run by the Legion implementation mechanism.
in response to createInstance() and createVectorInstance() calls on the class. To associate the implementation objects with their class, the tool simply calls the addImplementation() member function on the named class object, passing the name of the implementation object and its type. The class object then records this information in its local state, as shown in Figure 3.

At this point, when legion_create_implementation has been invoked six times, once for each of the three implementation types supported in both the Master and Slave classes, the two classes can now create instances of themselves. Moreover, these instances can be created on any SGI, Solaris, or Pentium host in Legion, including on hosts to which the programmer does not have direct access (i.e. a Unix account). The explicit step of installing the program on the site on which it will run does not have to be carried out by the programmer, and she need not even select the site if she doesn’t want to. This can be taken care of automatically by the implementation model, as described below.

### 4.3 Executing Programs

When the programmer wishes to run her code, she must invoke a function on Legion objects. This can be done from a Legion shell, which takes the name of the class of which to create an instance, or from within some other Legion object. In the Master / Slave example, the Master is created directly by the programmer from within a Legion shell, and the Master object is built to contain calls to create instances of the Slave class. We’ll deal with the case of creating Slave objects. To do so, the Master invokes the createVectorInstance() member function on the Slave class object. Note that when the application is programmed, the Master must be built to expect the existence of the Slave class at the time the Master is run. The parameters to createVectorInstance() include the number of objects to create, and a vector of placement decisions. This vector can be empty, leaving the decision completely up to the class object (which can then delegate it to a scheduling agent). At the other end of the spectrum, the vector can contain an explicit host for each object being created. Or, the vector can contain placement decisions for a subset of the objects. The full range of options is part of the scheduling model, and is described in more detail in [2].

Eventually, the scheduling vector will be filled with explicit hosts on which to place the objects, whether the vector is filled in by the caller of createVectorInstance() or by the class object or scheduling agent. Then, this placement decision is checked for consistency. For example, the set of hosts should not contain an implementation type for which the class does not have an appropriate implementation object. If it does, the createVectorInstance() call fails. (The set of supported implementation types is considered when the class or scheduling agent fills in the vector, but the vector is currently checked anyway.) Further, the selected hosts should all be willing to create instances of the class. That is, hosts implement their own security policy, and that policy should not preclude the creation of an instance of the Slave object type by the Master object. If it does, then the call fails and an error is returned to the Master, which invoked createVectorInstance(). [8].

Suppose now that the scheduling vector for the array of Slave objects contains eight Intel machines in a
cluster at one site, and four SGI and four Solaris machines in a different cluster. Once the scheduling vector is deemed appropriate, then the object creation begins. The Slave class steps through the vector and invokes the createObject() function on each host object named in the vector, passing the name of the appropriate implementation object for that host’s architecture as a parameter. (The class object can directly query a host object to determine its implementation type, and can maintain a local table of host to implementation type mappings.)

Upon receiving this call, each host then contacts its implementation cache object. The first time this is done for the Intel cluster, the implementation object will have to be downloaded by the implementation cache. The cache receives the name of the implementation object as a parameter from the host, creates a new local file in a directory that is shared by the hosts it is servicing, and reads the bytes from the implementation object and writes them into the file (Figure 4). The cache then makes an entry for this implementation object in its table, and returns the name of the file to the host object.

When the host receives this file, it executes it to create a new process. The local physical address of the process (for example, the IP address and port number pair) is returned to the class object, so that it can be handed out when other objects wish to bind to the newly created instance.

Subsequent calls to the createObject() function on hosts that share the same implementation cache within the Intel cluster can be services directly by the cache, since the name of the implementation object will match one in the cache. Thus, the cost of downloading the executable is paid once for the entire cluster. The other cluster (of Sun and SGI machines) will require two different implementation downloads, one for each implementation type, even though they share the same implementation cache.

5. Component-Based Implementations

The Legion implementation model also supports component-based application development. In contrast to some other implementations, components of a Legion object run within a single address space. When supporting objects built from components, rather than maintaining an implementation object per supported implementation type, a class object instead maintains sets of implementation component objects (ICOs), one set per type. Whereas implementation objects typically contain executables, implementation component objects maintain components of executables (e.g. Unix shared objects, DLLs, Java .jar files, etc.) that can be composed at runtime within the execution environment to represent the instance of a class. For example, a class might use a math library, and rather than compiling it statically into each executable, it can maintain a pointer to an ICO that is loaded into the executable at runtime. Instead of having to be installed on all platforms on which the object may run, the library need only be logically contained once in Legion (although replicants will exist for performance). The interface to ICOs matches the interface to implementation objects, allowing them to be downloaded and cached by implementation caches without any added support. The “constructor” for an object dynamically incorporates the components when the object is created or moves to a different host.

5.1 Instance Creation Using Components

The mechanism for creating objects comprised of implementation components contains an extra step beyond the mechanism for objects that are created using static monolithic executables. The first phase is similar: the class object, scheduling agent, host object, and implementation cache cooperate, as described above, to download and run an executable to represent the object. This executable, however, contains only a subset of the functions that instances of the object type should contain. At a minimum, this subset can contain only those functions that allow the object to receive and incorporate additional components. Once this “seed” executable is up and running, it then uses the implementation cache to download its implementation

---

1. An alternative view has components in separate independent address spaces. Legion supports this view by directly implementing each component as a different Legion object; therefore, support for this alternative view belongs on top of the Legion implementation model, rather than within it, and is outside the scope of this paper.
components by calling the getImplementation() function on the cache, once per component. The cache retrieves the components and stores them in the local file system, just as it does with implementation objects. The seed executable can then directly load the components into the running image of the executable, for example by utilizing runtime access to the Unix dynamic linker via the dlopen() function.

Legion’s mechanism for component-based object development can be used to support “dynamically configurable” objects, which can swap their implementation components in and out of objects on the fly, without requiring the objects to be shut down or redeployed. This functionality is described in detail in [10].

6. Summary

We have described the Legion implementation model, which defines how Legion creates processes to represent active objects. The model separates functionality into five different object types—class objects, scheduling agents, host objects, implementation objects, and implementation caches. Heterogeneity is addressed by defining the notion of an implementation type, which simultaneously describes executables (as abstracted by implementation objects) and execution platforms (as abstracted by host objects). Scheduling agents can use the implementation types to match executables with appropriate platforms at run-time. The four object types cooperate to download executables to appropriate execution platforms on demand, removing the requirements that programmers have accounts on all machines in the computational grid, and that they install their application on all sites on which it may run. Scalability is achieved (i) by distributing responsibility for maintaining implementations to application specific class objects, rather than maintaining it in a centralized location, and (ii) by defining the interface of implementation caches, which allow host objects to share the overhead of downloading executables, and to incur the downloading cost only once per object type.

7. References