Complete doc

Saturday, May 31, 2014 1:01 PM

9.4 THE LEGION GRID ARCHITECTURE

In the preceding sections we have outlined many of the technical problems that are associated with extending contemporary component and distributed-object technology to support grid applications. In the remainder of this chapter, we address the problem of delivering this type of programming infrastructure to the application builder. More specifically, the component architecture that the programmer uses is a high-level programming model, which must provide easy-to-use abstraction for complex grid services.

The task of implementing application-level programming abstractions in terms of basic grid functionality is a major challenge. There are three ways to address this problem. One approach, explored in Chapter 10, is to extend existing commodity technology. A second approach is to layer an application-level component architecture on top of a grid architecture such as the Globus toolkit, described in Chapter 11. The current versions of HPC++ and CC++, for example, use Nexus, the Globus communication layer, to support object-oriented RMI, and Java RMI has been ported to run over Nexus [80]. An effort to extend this object layer to a Globus-compatible component model is under way. The third approach is to provide a single, coherent virtual machine that addresses key grid issues such as scalability, programming ease, fault tolerance, security, and site autonomy completely within a reflective, object-based metasystem. The University of Virginia's Legion system is the best example of this type of grid architecture.

Legion is designed to support millions of hosts and trillions of objects existing in a loose confederation and tied together with high-speed links. The user can sit at a terminal and manipulate objects on several processors, but has the illusion of working on a single powerful computer. The objects the user manipulates can represent data resources, such as digital libraries and video streams; applications, such as teleconferencing and physical simulations; and physical devices, such as cameras, telescopes, and linear accelerators. Naturally, the objects being manipulated may be shared with other users. It is Legion's responsibility to support the abstractions presented to the user; to transparently schedule application components on processors; to manage data migration, caching, transfer, and coercion; to detect and manage faults; and to ensure that the user's data and physical resources are adequately protected.

9.4.1 Legion Design Objectives

The Legion design is based on 10 central objectives as follows.

1. *Site autonomy.* Legion will not be a monolithic system. It will be composed of resources owned and controlled by an array of organizations. These

1. *Site autonomy:* Legion will not be a monolithic system. It will be composed of resources owned and controlled by an array of organizations. These organizations, quite properly, will insist on having control over their own resources—for example, specifying how much of a resource can be used, who can use it, and when it can be used.
2. *Extensible core:* It is not possible to know or predict many current and future needs of all users. Legion's mechanism and policy must be realized via extensible and replaceable components that permit Legion to evolve over time and allow users to construct their own mechanisms and policies to meet their specific needs.
3. *Scalable architecture:* Because Legion will consist of millions of hosts, it must have a scalable architecture rather than centralized structure. This means that the system must be totally distributed.
4. *Easy-to-use, seamless computational environment:* Legion must mask the complexity of the hardware environment and of the communication and synchronization involved in parallel processing. Machine boundaries, for example, should be invisible to users, and compilers acting in concert with runtime facilities must manage the environment as much as possible.
5. *High performance via parallelism:* Legion must support easy-to-use parallel processing with large degrees of parallelism. This requirement includes task and data parallelism and their arbitrary combinations.
6. *Single persistent name space:* One of the most significant obstacles to wide area parallel processing is the lack of a single name space for file and data access. The existing multitude of disjoint name spaces makes writing applications that span sites extremely difficult.
7. *Security for users and resource owners:* Because Legion does not replace existing operating systems, we cannot significantly strengthen existing operating system protection and security mechanisms. In order to ensure that existing mechanisms are not weakened by Legion, it must provide mechanisms that allow users to manage their own security needs. Legion should not define the user's security policy or require a "trusted" Legion.
8. *Management and exploitation of resource heterogeneity:* Legion must support interoperability between heterogeneous hardware and software components, as well as take advantage of the fact that some architectures are better than others at executing particular applications (e.g., vectorizable codes).
9. *Multiple language support and interoperability:* Legion applications will be written in a variety of languages. It must be possible to integrate heterogeneous source-language application components in much the same manner that heterogeneous architectures are integrated. Interoperability requires that Legion support legacy codes.
10. *Fault tolerance:* In a system as large as Legion, it is certain that at any given instant several hosts, communication links, and disks will fail. Dealing with these failures and with the resulting dynamic reconfiguration is a necessity for both Legion and its applications.

[illegible] several hosts, communication links, and disks will fail. Dealing with these failures and with the resulting dynamic reconfiguration is a necessity for both Legion and its applications.

In addition, the Legion design is shaped by the following three constraints. First, Legion cannot replace host operating systems. Organizations will not permit their machines to be used if their operating systems must be replaced. Operating system replacement would require them to rewrite many of their applications, retrain many of their users, and possibly make their machines incompatible with other systems in their organization.

Second, Legion cannot legislate changes to the interconnection network, but must assume that the network resources and the protocols in use are outside any one group's control and should be accepted as an ungovernable element in large-scale parallel processing.

And, finally, Legion cannot insist that it be run as 'root' (or the equivalent). Indeed, quite the contrary: most Legion users will want it to run with the fewest possible privileges in order to protect themselves.

9.4.2 Legion System Architecture

Legion is a reflective object-based system that endows classes and metaclasses (classes whose instances are themselves classes) with system-level responsibility. Legion users will require a wide range of services on various levels, including security, performance, and functionality. No single policy or set of policies will satisfy every user; hence, whenever possible, users must be able to decide which trade-offs are necessary and desirable. Several characteristics of Legion's architecture reflect and support this philosophy.

Everything Is an Object

The Legion system consists of a variety of hardware and software resources, each of which is represented by a Legion object (defined as an active process that responds to member function invocations from other objects in the system). Legion describes the message format and high-level protocol for object interaction, but not the programming language or the communications protocol.

Classes Manage Their Instances

Every Legion object is defined and managed by its class object, which is itself an active Legion object. Class objects are given system-level responsibility: classes create new instances, schedule them for execution, activate and deactivate them, and provide information about their current location to client objects that wish to communicate with them. In this sense, classes act as managers and make policy, as well as define instances. Classes whose instances are themselves classes are called *metaclasses*.

objects that wish to communicate with them. In this sense, classes act as managers and make policy, as well as define instances. Classes whose instances are themselves classes are called *metaclasses*.

Users Can Provide Their Own Classes

Legion allows users to define and build their own class objects, which permits programmers to determine and even change the system-level mechanisms that support their objects. Legion 1.0 (and future Legion systems) contains default implementations of several useful types of classes and metaclasses. Users are not forced to use these implementations, however, particularly if the implementations do not meet the users' performance, security, or functionality requirements.

Core Objects Implement Common Services

Legion defines the interface and basic functionality of a set of core object types, which support basic system services such as naming and binding, and object creation, activation, deactivation, and deletion. Core Legion objects provide the mechanisms that classes use to implement policies appropriate for their instances. Examples of core objects include hosts, vaults, contexts, binding agents, and implementations.

9.4.3 The Legion Object Model

Legion objects are independent and logically address-space-disjoint active objects that communicate with one another via nonblocking method calls, which may be accepted in any order by the called object. Each method has a signature that describes the parameters and return value, if any, of the method. The complete set of method signatures for an object fully describes its interface (which is determined by its class). Legion class interfaces can be described in an IDL, several of which will be supported by Legion.

Naming System

Legion implements a three-level naming system. At the highest level, users refer to objects using human-readable strings, called *context names*. Context objects map context names to LOIDs (Legion object identifiers), which are location-independent identifiers. Each identifier includes an RSA public key. Since LOIDs are location independent, they are insufficient for communication by themselves. A LOID is therefore mapped to an LOA (Legion object address) for communication. An LOA is a physical address (or set of addresses, in the case of a replicated object) that contains sufficient information to allow other objects to find and communicate with the object, for example, an (IP address, port number) pair.

Object States

A Legion object can be in one of two different states, active or inert. As designed, Legion will contain too many objects for all to be represented simultaneously as active processes and therefore requires a strategy for maintaining

A Legion object can be in one of two different states, active or inert. As designed, Legion will contain too many objects for all to be represented simultaneously as active processes and therefore requires a strategy for maintaining and managing representations of these objects in their inert state in persistent storage. An inert object is represented by an object-persistent representation (OPR), which is a set of associated bytes residing in stable storage somewhere in the Legion system. The OPR contains information about an object's state that enables the object to move to an active state. An active object runs as a process that is ready to accept member function invocations; an active object's state is typically maintained in the address space of the process, although this is not strictly necessary.

Core Objects

Several core object types implement the basic system-level mechanisms required by all Legion objects. Like classes and metaclasses, core objects are replaceable system components; users (and in some cases resource controllers) can select or implement appropriate core objects.

Binding agents are Legion objects that map LOIDs to LOAs. A (LOID, LOA) pair is called a binding. Binding agents can cache bindings and organize themselves in hierarchies and software combining trees, in order to implement the binding mechanism in a scalable and efficient manner.

Context objects map context names to LOIDs, allowing users to name objects with arbitrary high-level string names, and enabling multiple disjoint name spaces to exist within Legion. All objects have a current context and a root context, which define parts of the name space in which context names are evaluated.

Host objects represent processors in Legion. One or more host objects run on each computing resource that is included in Legion. Host objects create and manage processes for active Legion objects. Classes invoke member functions on host objects in order to activate instances on the computing resources that the hosts represent. Representing computing resources with Legion objects abstracts the heterogeneity that results from different operating systems having different mechanisms for creating processes. Further, it provides resource owners with the ability to manage and control their resources as they see fit.

Just as the host object represents computing resources and maintains active Legion objects, the *vault object* represents persistent storage, but only for the purpose of maintaining the state, in OPRs, of the inert Legion objects supported by the vault.

Implementation objects allow Legion objects from other Legion systems to run as processes in the system. An implementation object typically contains machine code that is executed when a request to create or activate an object is made. More specifically, an implementation object is generally maintained as an executable file that a host object can execute when it receives a request to activate or create an object. An implementation object (or the name of an implementation object) is transferred from a class object to a host object to enable the host to create processes with the appropriate characteristics.

Legion specifies functionality and interfaces, not implementations. Legion 1.0 provides useful default implementations of class objects and of all the core

system objects, but users are never required to use the defaults. In particular, users can select (or build their own) class objects, which are empowered by the object model to select or implement system-level services. This feature of the system enables object services (e.g., creation, scheduling, security) to be made appropriate for the object types on which they operate, and eliminates Legion's dependence on a single implementation for its success.

9.5 A CLOSER LOOK AT LEGION

Space limitations do not permit a detailed discussion of how Legion realizes its objectives. Thus, rather than attempt to compress a large and complex system into a few pages, we will briefly expand on three aspects of Legion that are of interest to the high-performance computing community: security, high performance, and scheduling and resource management.

9.5.1 Security

Legion offers the opportunity of bringing the power and resources of millions of interlinked computers to the desktop computer. While that possibility is highly attractive, users will adopt Legion only if they feel confident that it will not compromise the privacy and integrity of their resources. Without security, Legion systems can offer some limited uses. But if the full Legion vision of a worldwide metacomputer is to become a reality, reliable and flexible security is essential.

Security Problems

Security has been a fundamental part of the Legion design from the beginning. Early work identified two main problems: users must be able to install Legion on their sites without significant risk, and they must be able to protect and control their Legion resources as they see fit.

The solution to the first problem is reflected in the broad design goals for Legion. Specifically, Legion does not require any special privileges from the host systems that run it. Administrators have the option of taking a very conservative approach while installing the system. Furthermore, Legion is defined as an architecture, not an implementation, allowing individual sites to reimplement functionality as necessary to reflect their particular security constraints.

The second problem, protecting Legion resources, requires multiple solutions. In an environment where users may range from students to banks to defense laboratories, it is impossible for Legion to dictate a single security policy that can hope to satisfy everyone. Therefore, Legion uses a flexible framework that adapts to many different needs. Individual users can choose how much they are willing to pay in time and convenience for the level of security they want. They can also customize their Legion system's security policies to match their organization's existing policies.

Placing policy in the hands of users is much more than just an attractive design feature. A decentralized system does not use security architectures based on control and mediation by "the system." Nor is there a single owner who sets and enforces global policies. In such an environment, users must ultimately take responsibility for security policies. Legion is designed to facilitate

The second problem, protecting Legion resources, requires multiple solutions. In an environment where users may range from students to banks to defense laboratories, it is impossible for Legion to dictate a single security policy that can hope to satisfy everyone. Therefore, Legion uses a flexible framework that adapts to many different needs. Individual users can choose how much they are willing to pay in time and convenience for the level of security they want. They can also customize their Legion system's security policies to match their organization's existing policies.

Placing policy in the hands of users is much more than just an attractive design feature. A decentralized system does not use security architectures based on control and mediation by "the system." Nor is there a single owner who sets and enforces global policies. In such an environment, users must ultimately take responsibility for security policies. Legion is designed to facilitate that goal.

The Security Model

The basic unit in Legion is the object, and the Legion security model is therefore oriented toward protecting both objects and object communication. Objects are accessed and manipulated via method calls; an object's rights are centered in its capabilities to make those calls. A file object may support methods for read, write, seek, and so forth, so that the read right for a file object might permit read and seek, but not write. The user determines the security policy for an object by defining the object's rights and the method calls they allow. Once this step is done, Legion provides the basic mechanism for enforcing that policy.

Every object in Legion supports a special member function called "MayI" (objects with no security have a NULL MayI). MayI is Legion's traffic cop: All method calls to an object must first pass through MayI before the target member function is invoked. If the caller has the appropriate rights for the target method, MayI allows that method invocation to proceed.

To make rights available to a potential caller, the owner of an object gives the caller a certificate listing the rights granted. This certificate cannot be forged. When the caller invokes a method on the object, it presents the appropriate certificate to MayI, which then checks the scope and authenticity of the certificate. Alternatively, the owner of an object can permanently assign a set of rights to a particular caller or group. In that case, MayI's responsibility is to confirm the caller's identity and membership in one of the allowed groups and then to compare the rights authorized with the rights required for the method call.

Besides regulating user access control, Legion also protects underlying communications between objects. Every Legion object has a public-key pair; the public key is part of the object's name (its LOID). Objects can use the public key of a target object to encrypt their communications to it. Likewise, an object's private key can be used to sign messages, thereby providing authentication and nonrepudiation. This integration of public keys and object names eliminates the need for a certification authority. If an intruder tries to tamper with the public key of a known object, the intruder will create a new and unknown name.

The combined components of the security model encourage the creation of a large-scale Legion system with multiple overlapping trust domains. Each domain can be separately defined and controlled by the users that it affects. When difficult problems arise, such as merging two trust domains, Legion provides a common and flexible context in which they can be resolved.

domain can be separately defined and controlled by the users that it affects. When difficult problems arise, such as merging two trust domains, Legion provides a common and flexible context in which they can be resolved.

9.5.2 High Performance

Legion achieves high-performance computing in two ways: by selecting processing resources based on load and job affinity, and by parallel processing.

Even single-task jobs can have better performance when presented with a range of possible execution sites. The user can choose the host with the lowest load or the greatest power. Power, in this context, might be defined by performance on the SPEC benchmarks adjusted for load, or by using the application itself as a benchmark. Similarly, different components of a coarse-grained meta-application may be scheduled on different hosts (based on the component's affinity to that type of host), leading to a phenomenon known as *superconcurrency* [214]. In either scenario, Legion's flexible resource management scheme lets user-level scheduling agents choose the right resource.

Alternatively, Legion can be used for traditional parallel processing, as when executing a single application across geographically separate hosts, or supporting meta-applications (e.g., scheduling the components of a single meta-application on the nodes of an MPP). Legion supports a distributed-memory parallel computing model in four ways: supporting parallel libraries, supporting parallel languages, wrapping parallel components, and exporting the runtime library interface.

Supporting Parallel Libraries

The vast majority of parallel applications written today use MPI [250] or PVM [227]. Legion supports both MPI's and PVM's libraries via emulation libraries, which use the underlying Legion runtime library. Existing applications need only to be recompiled and relinked in order to run on Legion.

Supporting Parallel Languages

Legion supports MPL (Mentat Programming Language, described in [569]), BFS (Basic Fortran Support), and Java. MPL is a parallel C++ language in which the user specifies those classes that are computationally complex enough to warrant parallel execution. Class instances are then used like C++ class instances: the compiler and runtime system take over, construct parallel computation graphs of the program, and then execute the methods in parallel on different processors. Legion is written in MPL. BFS is a set of pseudocomments for Fortran and a preprocessor that gives the Fortran programmer access to Legion objects. It also allows parallel execution via remote asynchronous procedure calls, as well as the construction of program graphs. The Java interface allows Java programs to access Legion objects and to execute member functions asynchronously.

Wrapping Parallel Components

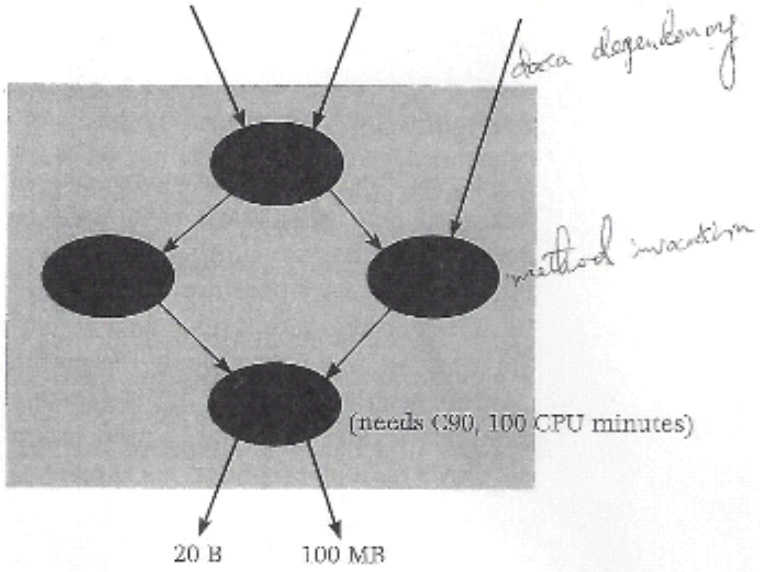
Object wrapping is a time-honored tradition in the object-oriented world, but Legion extends the notion of encapsulating existing legacy codes into objects one step further by encapsulating a parallel component into an object. To other Legion objects, the encapsulated object appears sequential, but executes faster. Thus, one could encapsulate a PVM, HPF, or shared-memory threaded application in a Legion object.

faster. Thus, one could encapsulate a PVM, HPF, or shared-memory threaded application in a Legion object.

Exporting the Runtime Library Interface

The Legion team cannot provide the full range of languages and tools that users need. The designers of Legion, rather than developing everything at the University of Virginia, intended the system to be an open community artifact to which other languages and tools are ported. To support third-party software development, the complete runtime library interface is available and may be directly manipulated by user libraries. The Legion library is completely reconfigurable: It supports basic communication, encryption/decryption, authentication, exception detection and propagation, and other features. One feature of particular interest is program graph support.

Program graphs (Figure 9.6) represent functions and are first class and recursive. Graph nodes are member function invocations on Legion objects or subgraphs. Arcs model data dependencies. Graphs are constructed by starting



9.6 FIGURE A Legion program graph: functions of three arguments and two outputs.

with an empty graph and adding nodes and arcs. Graphs may be combined, resulting in a form of function composition. Finally, graphs may be annotated with arbitrary information, such as resource requirements and architecture affinities. The annotations may be used by schedulers, fault tolerance protocols, and other user-defined services.

9.5.3 Scheduling and Resource Management

The Legion scheduling philosophy is one of reservation through a negotiation process between resource providers and resource consumers. Autonomy is considered to be the single most crucial aspect of this process, for two reasons.

First, site autonomy is crucial in attracting resource providers. In particular, participating sites must be assured that their local policies will be respected by the system at large. Therefore, final authority over the use of a resource is placed with the resource itself.

lar, participating sites must be assured that their local policies will be respected by the system at large. Therefore, final authority over the use of a resource is placed with the resource itself.

Second, user autonomy is crucial to achieving maximum performance. A single scheduling policy will not be the best answer for all problems and programs: Users should be able to choose between scheduling policies, selecting the one that best fits the problem at hand or, if necessary, providing their

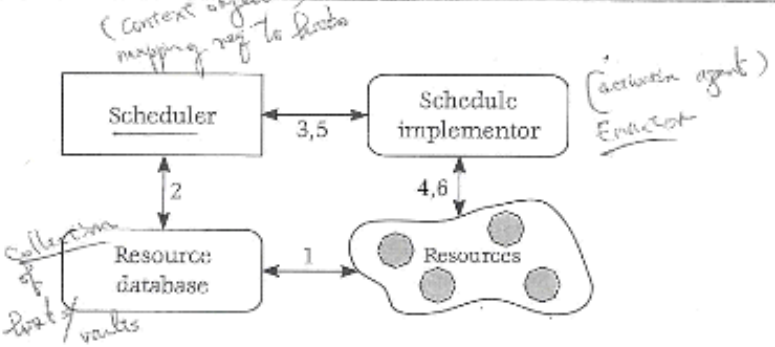


FIGURE 9.7 The Legion scheduling model.

own schedulers. A special, and vitally important, example of user-provided schedulers is that of application-level scheduling. This allows users to provide per-application schedulers that are specially tailored to match the needs of the application. Application-level schedulers will be commonplace in high-performance computing domains.

Legion currently provides two types of resources: computational resources (hosts) and storage resources (vaults). Network resources will be incorporated in the future. As seen in Figure 9.7, the Legion scheduling module consists of three major components: a resource state information database, a module that computes request mapping to resources (hosts and vaults), and an activation agent responsible for implementing the computed schedule. These items are called the Collection, Scheduler, and Enactor, respectively.

The Collection interacts with resource objects to collect information describing the system's state (Figure 9.7, step 1). The Scheduler queries the Collection to determine a set of available resources that match the Scheduler's requirements (step 2). After computing a schedule, or set of desired schedules, the Scheduler passes a list of schedules to the Enactor for implementation (step 3). The Enactor then makes reservations with the individual resources (step 4) and reports the results to the Scheduler (step 5). Upon approval by the Scheduler, the Enactor places objects on the hosts and monitors their status (step 6).

If the user does not wish to select or provide an external scheduler, the Legion system (via the class mechanism) provides default scheduling behavior that supplies general-purpose support. Through the use of class defaults, sample schedulers, and application-level schedulers, the user can balance the effort put into scheduling against the resulting application performance gain.

9.6 APPLICATION SCENARIOS AND LEGION

To conclude this overview of Legion, let us revisit the application scenarios described in the first half of this chapter. A Legion implementation of the teleimmersion collaboration design application would closely follow the general object-oriented design presented earlier. The display object, the visual database object, the simulation components, the design database, and the user interface control objects would all be Legion objects that would communicate via method invocations.

The Digital Sky project is a more interesting example of a system that can exploit Legion. In a Legion implementation, the application "object databases" that contain the observations would be Legion objects—perhaps instances of an `observation_db` class. An `observation_db` object would have an interface tailored to the application. Thus, rather than generic (and therefore hard to optimize for the application) functions such as `read()` and `write()` or `select-from-where()`, an `observation_db` object would have functions such as `get_sky_volume()` or `get_object()`. In fact, the interface is completely arbitrary, allowing the designer the choice of query and update interfaces.

Access to `observation_db` instances would be location transparent because of the nature of Legion LOIDs. They could also be online versus archive transparent. That is, the Legion vault storing the persistent state of the objects could mask whether the state is on disk or stored on an archival medium such as tape.

The implementation of `observation_db` could then be optimized for the type of data stored and the most common data requests. For example, high performance for large sparse databases can be realized by using a PLOP file [502] or a quad-tree [487], as has been done for radio astronomy data [303, 302]. Data could also be previously fetched and cached based on access predictions provided by the user via special member functions, rather than using a demand-driven strategy and naive assumptions about temporal and spatial locality, as is typically the case.

The internal implementation of `observation_db` could be internally parallel as well. The data could be horizontally partitioned across multiple subobjects, each of which resided on a separate device. Queries against the `observation_db` could then be executed in parallel, with multiple devices active at the same time, resulting in greater bandwidth.

Legion further supports the Digital Sky requirements by providing the following:

- A flexible access control policy that can easily be tailored to meet a variety of needs
- Support for flow-oriented processing via MPL, BFS, and the underlying graph support mechanism
- Support for user-level scheduling that allows either the computation to be moved to the data or the data to be moved to the computation
- The ability to dynamically insert object-monitoring code for performance debugging
- The ability to encapsulate legacy databases in Legion objects by wrapping them in an object and restricting the object's placement to the particular host or hosts where the legacy system can run

Finally, using Legion's unique metaclass system, one could create a metaclass for the `observation_db` class that supported object replicas. Replicas could be generated and placed as needed at multiple sites for both faster access and increased availability in the event of equipment failure. Replicas could also be transparently generated on demand close to a data consumer, acting as intelligent prefetching and caching agents.

ACKNOWLEDGMENTS

Fritz Knabe, Steve Chapin, and Mike Lewis assisted in preparing the Legion material. Portions of the Legion material have appeared elsewhere, specifically the 10 design objectives and the three constraints, which have appeared in many Legion papers. The Legion work has been supported by the following grants and contracts: DARPA (Navy) contract no. N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, DOE D459000-16-3C, DARPA (GA) SC H607305A, and Northrop-Grumman.

FURTHER READING

For more information on the topics covered in this chapter, see *www.mkp.com/grids* and also the following references: