Legion: The Next Logical Step Toward a Nationwide Virtual Computer

Andrew S. Grimshaw
William A. Wulf
James C. French
Alfred C. Weaver
Paul F. Reynolds Jr.

Technical Report No. CS-94-21
June, 1994

Legion: The Next Logical Step Toward a Nationwide Virtual Computer

e pluribus unum -- one out of many

Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, Paul F. Reynolds
Department of Computer Science, University of Virginia

Abstract

The coming of giga-bit networks makes possible the realization of a single nationwide virtual computer comprised of a variety of geographically distributed high-performance machines and workstations. To realize the potential that the physical infrastructure provides, software must be developed that is easy to use, supports large degrees of parallelism in applications code, and manages the complexity of the underlying physical system for the user. This paper describes our approach to constructing and exploiting such "metasystems". Our approach inherits features of earlier work on parallel processing systems and heterogeneous distributed computing systems. In particular, we are building on Mentat, an object-oriented parallel processing system developed at the University of Virginia. This report is a preliminary document. We expect changes to occur as the architecture and design of the system mature.

1.0 Introduction

The information superhighway is upon us – it will provide the communication infrastructure for as yet undreamed of applications. But what will the highway connect? Will it connect separate islands of computational service, allowing them to no more than exchange information? Or will it allow integration of a multitude of islands into a single monolithic virtual machine – one machine out of many. Our goal is no less than the realization of such a virtual machine by combining the emergent opportunities provided by the NII and the computational and data resources already available into a single nation-wide metasystem, to create *Legion*, one out of many.

Legion will consist of workstations, vector supercomputers, and parallel supercomputers connected by local area networks, enterprise-wide networks, and the national information infrastructure. The potential aggregate computation power of such an assembly of machines is enormous, approaching a petaflop; this massive potential is, as yet, unrealized. These machines are currently tied together in a loose confederation of shared communication resources used primarily to support electronic mail, file transfer, and remote login. However, these resources could be used to provide far more than just communication services and electronic mail; they have the potential to provide a single, seamless, computational environment in which cycles, communication, and data are shared, and in which the workstation across the continent is no less a resource than one down the hall. Among the many benefits such an environment will make possible, three may be particularly critical: (1) increased productivity due to increased resource availability for researchers, (2) reduced gross computation resource requirements due to more efficient utilization of resources already available, and, (3) the ability to do larger problems than is currently possible.

Our objective is design and implement a prototype metasystem, Legion, that realizes the potential offered by the available resources. Legion will provide a single, seamless interface to users that provides large amounts of compute power, facilitates collaboration between users at different locations, and hides from the user the fact that the system is composed of hundreds to thousands of processors scattered across the country. Our vision is of a system in which a user sits at a Legion workstation and has the illusion of a single very powerful computer on her desk. The user sits down at a terminal[1] and invokes an application on a data set. It is then the responsibility of the metasystem to *transparently* schedule application components on processors, manage data transfer and coercion, and manage communication and synchronization in such a manner as to minimize[2] execution time via parallel execution of the application components. System boundaries will be invisible, as will the location of data and the existence of faults. In addition to the interface presented to the single user, Legion will provide an ideal platform upon which to construct collaboration systems, further enhancing system utility.

Before the Legion vision can be realized, several technical challenges must be overcome. These are software problems; the hardware challenges are being addressed and are the enabling technologies that provide the opportunity. The software challenges revolve around eight central themes: *achieving high performance via parallelism, managing and exploiting component heterogeneity, resourc*e *management, file and data access, fault-tolerance, ease-of-use and user interfaces, protection and authentication*, *and exploitation of high-performance protocols*. We realize that these are serious, non-trivial, issues; we examine the challenges in more detail in section 3.0.

In addition to the purely technical issues, there are political, sociological, and economic ones. These include encouraging the participation of resource-rich centers and the avoidance of the human tendency to free-ride. We must avoid the modern day version of the tragedy of the commons in which everyone consumes resources but does not contribute. We intend to prevent such practices by developing and employing accounting policies that encourage good community behavior.

The vision of a seamless metacomputer is not novel. Indeed, a number of systems have been designed to attack one or more of the problems mentioned above, e.g., Andrew, Locus and NSF for file systems [25][28][35], Locus for fault-tolerance, Sun XDR and the University of Washington HCS for heterogeneity[31][32][33]. None has been fully successful. What has changed that makes the realization of a complete high performance metacomputer possible? What has changed is that solutions to the problem of providing high-performance via parallelism, previously available only for tightly coupled parallel processors, are now available for loosely coupled distributed systems [1][3][4][8][9][12][21][26][29][36]. This change in the technological landscape makes possible the realization of practical high performance metasystems.

Whether or not a metasystem is explicitly constructed by design, the nation (and probably at some time the world) will eventually end up with a system that shares at least some of the attributes of Legion. The reason is simple, both individual and organizational users will be required to deal with increasingly obvious shortcomings of the nationwide computer system. Evidence exists already, in the appearance of, for example, worldwide web and gopher, that tools will be developed in an ad hoc fashion to paper over the

[1] We use terminal in its most liberal sense. Terminals may include any human I/O device, glass tty, X interfaces, or virtual reality interfaces such as head-mounted displays and data gloves.
[2] In general this is NP-hard. We really mean “do a good job” using a heuristic.

gaps between local systems. Other responses to difficult complexities, such as workstation farms [23] are already being developed (or are simply waiting to be developed).

The issue is not whether metasystems will be developed; they clearly will. It is, rather, whether they will come about by design which and in a coherent, seamless system--or painfully and in an ad hoc manner by patching together congeries of independently developed systems, each with different objectives, design philosophies, and computation models. In order to facilitate the design and construction of such metasystems, software infrastructures must be provided as foundations for applications.

If, however, a metasystem infrastructure is already in place, the potential benefits are enormous. The benefits we envision include: (1) more effective collaboration by virtually putting coworkers in the same workplace; (2) higher application performance due to parallel execution and exploitation of off-site resources; (3) improved access to data and computational resources for smaller sites; (4) improved researcher and user productivity resulting from the more effective collaboration and better application performance; (5) increased resource utilization; and (6) a considerably simpler programming environment for the applications programmers writing applications. Indeed, it seems probable to us that the NII can reach its full potential only with the Legion infrastructure (or some equivalent). In addition to these already visible benefits there will be as yet undreamt of applications and services once Legion is up and running.

The remainder of the report is in four sections. In the next section we spell out the objectives of the Legion project in detail, and then discuss our approach to achieving the objectives. In section 3 we examine the technical challenges to realizing a seamless metasystem, with particular attention to the system architecture that we intend to employ. We conclude with a discussion of the status of the project, and our next steps.

2.0 Legion

From our Legion vision we have distilled six primary design objectives that are central to the success of the project:

- Easy to use, seamless computational environment.
- High performance via parallelism.
- Single, persistent namespace.
- Security for both users and resource providers.
- Exploit heterogeneity for high performance.
- Minimize impact on resource owner’s local computation.

Easy to use, seamless computational environment. Legion must be easy to use or it won’t be used except by the most dedicated users. Legion must mask the complexity of the hardware environment and the complexity of communication and synchronization of parallel processing. Machine boundaries should be invisible to users. The user should see a single consistent name space. Remote login, ftp, uucp, will be demons from the past that will no longer torment users.Further, it should provide the user and programmer with a uniform interface to service. Otherwise the user may be overwhelmed with the different choices and mechanisms needed to accomplish applications tasks. These requirements argue for high-level language interfaces. We feel that asking programmers to write applications in such a wide-area, heterogeneous environment using low-level constructs such as send and receive would be to invite failure. It will be far too complex for all but the best of programmers. Therefore, as much as possible, compilers, acting in concert with run-time facilities, must manage the environment for the user.

High performance via parallelism. Legion must support easy-to-use parallel processing with large degrees of parallelism. This includes both task and data parallelism, as well as combinations of both. While it is true that many application component kernels are primarily data parallel, many applications consist of multiple communicating components. Further, because of the nature of the interconnection network, the system must be latency tolerant, and be capable of managing hundreds to thousands of processors. This implies that the underlying computation model and programming paradigms must be scalable.

Single, persistent namespace. 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. Therefore, Legion must provide a single name space for persistent objects (nee files) to applications programs.

Security for both users and resource owners. Because of the constraint that we cannot replace existing host operating systems (see below), we cannot strengthen the existing protection and security mechanism. Existing operating system protection and security mechanisms have known weaknesses. Strengthening operating system security is not a goal of our project. That said however, we must ensure that the existing mechanism is not weakened, and that Legion does not introduce any new holes into the sieve.

Exploit heterogeneity for high performance. Clearly Legion must accommodate heterogeneity, i.e., support interoperability between heterogeneous components. Legion will do more though, it will be able to exploit the diversity of hardware and data resources available in a metasystem, executing sub tasks of large applications on different heterogeneous processors, and using heterogeneous data sources. Some architectures are better than others at executing particular kinds of code, e.g., vectorizable codes. These affinities, and the costs of exploiting them, must be factored into scheduling decisions and policies.

Minimize impact on resource owner's local computation. The *noticeable* impact of Legion on local resources must be minimized, particularly on interactive sessions. If users notice a significant performance penalty when their site is attached to Legion they will resist. Any observed penalty must be more than offset by the benefits of Legionnaire status.

We have the additional non-design objective of demonstrating the effectiveness of wide-area heterogeneous computing on non-toy applications drawn from a variety of application domains. The applications will be drawn both from the "grand challenges", e.g., global climate modeling and the human genome project, and from economically significant (but not grand challenge) problem areas such as electrical engineering and medicine.

As well as the design goals above, our design is restricted by two constraints, we cannot replace host operating systems and we cannot legislate changes to the interconnection network.

We cannot replace host operating systems. This restriction is required for two reasons. First, 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 them incompatible with other systems in their organization. Our experience with Mentat indicates that it is sufficient to layer a system on top of an existing host operating system.

We cannot legislate changes to the interconnection network. We must initially assume that the network resources, and the protocols in use, are a given. Much as we must accommodate operating system heterogeneity, we must live with the available system. That is not to say though, that we cannot layer better protocols upon existing protocols, or that we cannot state that performance for a particular application on a particular network will be poor unless the protocol is changed.

2.1 Approach

The principles of the object-oriented paradigm constitute the foundation for the construction of Legion; our goal will be exploitation of the paradigm's encapsulation and inheritance properties. Use of an object-oriented foundation will make accessible a variety of the benefits often associated with it, including, software reuse, fault containment, and reduction in complexity. The need for the paradigm is particularly acute in a system as large and complex as Legion. Other investigators have proposed constructing application libraries and applications for wide-area parallel processing using only low-level message passing services such as those provided by PVM [34] and P4[3]. Use of such tools requires the programmer to address the full complexity of the environment; the difficult problems of managing faults, scheduling, load balancing, etc., are likely to overwhelm all but the best programmers. The consequences will be libraries and applications that either don't exploit the environment or that are failure prone. Moreover, why should be applications programmer need to be both an application domain expert and a fault-tolerant, high-performance parallel computing expert – re-inventing tools for every application? System software should be written once – freeing applications writers to concentrate on their specialties.

Objects, written in either an object-oriented language or other languages such as HPF Fortran, will encapsulate their implementation, data structures, and parallelism, and will interact with other objects via well-defined interfaces. In addition they may also have associated inherited timing, fault, persistence, priority, and protection characteristics. Naturally these may be overloaded to provide different functionality on a class by class basis. Similarly, a class may have multiple implementations with the same interface.

We are firmly committed to the object-oriented paradigm. At the same time we recognize that other languages and paradigms, in particular Fortran and its parallel dialects, HPF Fortran [26], Fortran 90, and Fortran D [9], have and will continue to have important roles. Legion will support those languages in two ways, first by supporting the encapsulation of programs written in other languages into objects, and second, by providing a direct interface to the Legion run-time system for compiler writers.

Our approach is evolutionary rather than revolutionary. To realize our vision of a seamless metasystem we have chosen to begin by first constructing a testbed by extending Mentat, an existing object-oriented parallel processing system [19]. Mentat attacks the problem of providing easy-to-use high performance parallelism to users. The performance aspects of Mentat have been demonstrated on several real-world applications, on hardware platforms spanning the bandwidth/latency space, and in a heterogeneous environment [15][16][18]. Mentat's ability to achieve high-performance on platforms with very different communications characteristics is the key factor in our choice of Mentat as our implementation vehicle.

The Mentat testbed provides us with an ideal platform to rapidly prototype and try out ideas, forcing the details and hidden assumptions to be carefully examined, and exposing flaws in the ideas or in the system components. There are two principal reasons for extending an existing system rather than starting work on Legion from scratch. First, building on Mentat will allow very substantial savings in the amount of code writing which will be required before initial applications can be executed. Second, we will be able to work with a system which we know works and with which we already have had very considerable experience. New capabilities can be incrementally added as new problems are addressed and their solutions incorporated. Using as existing system as a starting point may modestly constrain design choices and somewhat limit innovation; the use of a familiar and tested prototype will enhance our understanding of the problems inevitably to be encountered in developing a metasystem.

Our model for evolution of Legion is that of the ARPA-net. We will begin with a campus-wide virtual computer here on our own campus, then expand to a small community of participating sites. Legion will be an open system, rather than an exclusive club.

3.0 Technical Challenges

We next address the major issues in constructing Legion and briefly how we intend to address each one. We begin by first presenting the Legion system architecture followed by the major issues. Because of space limitations we only touch on each area rather than give it the full treatment it deserves. Because it is central the system architecture receives more attention.

3.1 System Architecture

The system architecture for a system such as Legion is one of its most critical components. A well thought out and elegant system architecture increases the likelihood that the resulting system will be responsive to the inevitable future changes and robust. Key aspects of our system architecture are: it is object-based; objects have a class and communicate via typed messages; it is implemented as a layered virtual machine; and it uses a scalable, distributed control.

Object-based: The primitive units are objects which invoke methods on one another. Legion objects have an address space, a class, a name, and a set of capabilities.

Legion objects are *independent* objects that contain *contained* objects. Independent objects are recognized by the system and are address space disjoint. Contained objects are language defined objects, e.g., integers or complex numbers, that are contained in the address space of an independent object. The system architecture does not prescribe how programing languages define, name, or manipulate contained objects, it is concerned only with independent objects and their interaction. Henceforth when we say object we mean independent object unless stated otherwise.

Object methods: Objects communicate via methods. Each method has a formal parameter list of zero or more typed arguments, and may return one typed value. Each formal parameter may be either "in", "out", or “in/out”. The default is “in”. All arguments are passed by value, i.e., the actual parameter is copied from the caller to the callee. All “out” parameters and the return value of the method are similarly copied by value to the consumer of the result (if any). The set of methods available for an object is defined by its class. The semantics of method invocation are asynchronous. The arguments are copied at the instant of invocation, and the return value(s), are available when the invoked method returns them. The caller may choose whether to block waiting for the results or to proceed until the results are needed.

Object methods may be accepted by the callee in any order that the callee chooses. The system will support an option to allow the callee to accept invocations in the order they were called by any particular caller. Thus, method invocation at the callee may be deferred until an earlier invocation has completed. It is important to note that the ordering of invocations from two different callers is not necessarily preserved.

Classes: All objects are instances of a class. Classes definitions are themselves instances of the class legion_class. A class definition consists of several components, an interface, one or more source implementations, one or more concrete implementations[3], one or more generators (a generator takes an appropriate textual implementation and constructs an executable for a particular platform), an owner, resource requirement information, and accounting information. In addition, like objects, classes have methods to instantiate class instances, build concrete implementations for a particular architectures, return concrete implementations (i.e., executables for a particular platform), update the source implementations, update the interface, add generators, etc.

Class interfaces: The class interface specifies the methods that can be invoked on a class instance. Each method has a name, and one or more typed formal parameters. The method name and the types of the formal parameters are combined to form a function signature as in C++. Thus, methods may be overloaded. The parameter types can be used by the system to provide type checking and data coercion. The interfaces will be specified in an interface description language (IDL). Interfaces for other languages, e.g., C++ or Ada, can be generated from the class IDL description. Conversely, we expect the IDL description to be generated by the various language compilers.

If instances of a class require persistent state we say that they are *stateful*, otherwise they are *stateless*. Stateful objects maintain state information from the instant the are instantiated to the instant of their destruction. Stateless objects may have state but their correct operation must not depend on the state persisting from one method invocation to the next. Thus, stateless objects should be thought of as pure functions. Further, successive invocations on the “same” stateless object are not guaranteed to be executed by the same instance. The state requirements of a class are a part of its interface.

It is not the case that all objects will be written in an object-oriented or object-based language. The class definitions for non-object based languages, e.g., Fortran, must be generated by hand and placed into the class description database.

Class sources (a.k.a. abstract implementations): There may be one or more implementations of a class. For example, the class may have an implementation in C++ for scalar architectures, and an implementation

[3] The concept of multiple concrete implementations for a single interface is borrowed from Emerald.

in C* for use on data parallel machines. The sources are used in combination with the generators to create concrete implementations (executables) for the class.

Concrete implementations: In order to run on a particular architecture a class must have a concrete implementation for that architecture. The concrete implementation is the executable.

Generators: A generator is a program that takes as input source file and produces a concrete implementation. In practice a generator may be nothing more than a makefile tuned to a particular architecture. Generators are used by the system to generate concrete implementations from the sources. The base class legion_class defines generators for each architecture. Classes may overload the default generators and define their own.

Class management and the class description database: The class description database (CDD) will contain information on all of the above attributes of a class. In addition the CDD will contain information on any special attributes of the class such as resource requirements (e.g., memory, scratch), processor affinity, processor requirements, and where the executables can be found. In addition, optional information such as author contact information and license and distribution information can be kept in the CDD. Further, the CDD will be accessible to tools such class browsers.

Naming: In Mentat names are addresses, they contain enough operating system dependent information to communicate with the named object using host operating system IPC facilities, e.g., a hostname and port pair. Such a naming scheme is inappropriate for Legion. Legion names do not contain the address of an object. Instead they are 128-bit globally unique identifiers. A name is bound to one or more addresses by the communication system. These bindings may change over time, if for example, the named object moves, or the membership in a named group of objects changes. It is the responsibility of the communication system, not user objects, to maintain binding information.

Thus far we have concentrated on the features of the system architecture from the programmers point of view rather than the implementation. The programmer views Legion from the point of view of services and, at the lowest level, objects as UID's. We next examine the implementation of the system architecture.

Layered Virtual machine: Legion will be constructed using a classic layered virtual machine design in which successive layers of the machine are implemented using lower level services. Architectural differences will be isolated as much as possible in low-level, machine dependent, modules. All other modules will be architecture independent.

The lowest level consists of object management services, kill, create, low and mid-level scheduling; and communication. Object management services are implemented using host operating system services, e.g., fork, exec, read, and write. Basic communication services are implemented using operating system calls as well, e.g., sockets, send, or MPI libraries. The higher level services are not yet completely defined. They include a high-level scheduler, a name service, the fault management system, macro dataflow machine services, data coercion, encryption/decryption, and an extensible file system [20].

The choice of a layered virtual machine has performance implications. The induced overhead will reduce application performance below what is possible writing each application and high level service on the bare machine. However, software complexity and effort are significantly reduced. Our experience with Mentat (which uses a layered virtual machine) is that the performance "hit" is acceptable and well worth the trade-off in software complexity. Indeed, for a system such as Legion we do not believe that it is reasonable to use any other approach.

The Object Management System (OMS): The object management system is responsible for implementing the object model presented to the programmer in which potentially hundreds of thousands of objects, each with its own thread of control and implementation, name one another with UID's, communicate via method invocation, may migrate, and "live" on potentially faulty hosts. Thus the OMS is responsible for object scheduling, object migration, maintaining bindings from UID's to object addresses, and fault-tolerance. Rather than present the complete design (which is still evolving) we present two components of the design, the implementation of hundreds of thousands of objects using limited processing resources, and naming and binding.

We expect that there will be insufficient resources in the system to implement every Legion object as a process. There will be too many Legion objects. Therefore, in our implementation an object may be *active* or *inactive* at any given instant in time. An active object has an in-memory representation somewhere, i.e. it has an address space and a thread of control. An inactive object is not currently running anywhere, it cannot receive or process messages. Whether an object is active or inactive is irrelevant to users or other objects. The system is responsible for delivering messages to an object regardless of whether is it active or not.

The OMS will maintain a database of object information. For each object the OMS will keep track of its state (active or inactive), if active, its address (where it is running), and the location on stable store where its persistent state is stored.

In order for the OMS to activate and deactivate an object the object's class must have the methods *capture_state*() and *restore_from_state*() defined. To deactivate an object the OMS will capture its state, save the state to stable store (disk), update the OMS database, and release the process resources held by the object. Similarly, to activate an object the OMS will find a suitable location (a scheduling decision), allocate process resources, begin execution of the appropriate concrete implementation, read the state from stable store, invoke the *restore_from_state*() method, and update the OMS database.

The capture/restore state mechanism can be used not only to multiplex objects onto limited process resources, it can also be used to provide processes migration by saving the state on one processor and restoring it on another, and for simple fault-tolerance by periodically saving the state of an object, and restoring the object to it's last checkpoint.

Naming and binding: The naming scheme is a two layer scheme. Symbolic user names (strings) are mapped to UID's by name servers which are themselves objects. We explicitly allow multiple name servers, and users to define their own name servers. There is one well-known name server class that provides basic name service and is used as a bootstrap name server.

Objects communicate with other objects via method invocation. Methods are invoked on the UID's. The

OMS maintains mappings from UID's to object addresses for active objects. In order to avoid repeated queries by objects to the OMS for UID bindings, each object will maintain a local cache of UID to address bindings. When a method is invoked on an object the cache is checked for the UID. If the address is in the cache it is used. If not the OMS is queried to determine the address, and the address is added to the local cache. When objects change their address (via migration or failure), the caches are invalidated. Note the similarity to address resolution in a segmented memory system, in particular the named object(s) may not be active. They may need to be brought into memory by OMS in order to receive a message.

3.2 Achieving high-performance via parallelism

One of the primary motivations for constructing Legion is performance. If higher application performance cannot be achieved using Legion than could be achieved locally at each site, then a rational site would choose not to become part of Legion. High performance for applications can be achieved in two ways, by using the faster sequential machines available at other sites, or by running the application in parallel across the components of Legion. If the first approach is used exclusively, then those sites with the faster machines will soon tire of providing cycles for everyone else.

An application will not become a parallel application simply by executing in Legion, it must first be parallelized. We intend to support parallel execution in Legion using three mechanisms, providing a basic invocation and object management API to programmers, exposing the Legion run-time interface to parallel language compilers, and by providing a shell-like meta-language that executes applications in parallel when data dependencies permit.

Not all applications will benefit from parallel execution under Legion. As is true in any parallel processing system there will be applications that either do not parallelize well, or are too fine-grain for the environment. Applications that will perform well under Legion will be latency tolerant, and relatively large-grain. An exception being application components written in other languages for a specific machine, e.g., a C* application component running on a CM-5 (see section 3.4). There are, however, a large number of applications that are large-grain and latency tolerant.

A final aspect of our approach to parallelism is *resource transparency.* By resource transparency we mean that the application should not depend on a particular number and type of processors being allocated to it for execution. Instead we expect that the resources available for a particular application will vary from run to run. It should not be necessary to recompile the application.

Message passing API: A large number of parallel applications have been written using low-level message passing. While we do not believe that this is the right paradigm for the construction of large software systems we recognize that we must support these applications, at least in transition. Therefore we will provide several message passing compatibility libraries to programmers (PVM [34], P4 [3]), and will cooperate with commercial vendors (Express^{TM}, Linda [10]) to port their libraries.

Exposing the Legion run-time: Legion will be an open system in order to encourage third party software development. We will expose the Legion run-time to compiler writers to permit third party language providers (e.g., HPF Fortran, DataParallel C, C*, pC++, ADA) to port their compilers to Legion. Several parallel languages use one of the standard message passing API's already as their message fabric, e.g., DataParallel C, Paragon. Those languages will be able to transition rapidly using Legion provided API's.

We will also support the Mentat programming language (MPL) with Legion from the very beginning. MPL is an extension of C++ designed to support the parallel execution of applications. Several real applications have been developed using MPL. These applications will be used to test the efficacy of our approach from a very early stage.

Parallel shell: Our final mechanism is the development of a parallel shell similar to DQS and other network queueing system [23]. Applications are executed in parallel, rather than sequentially, whenever data dependencies permit[4]. Data dependencies exist when an application generates an output file that is later used as an input to another application. This is a relatively simple technology that permits the many applications that are essentially shell scripts, to be parallelized.

3.3 Managing and exploiting component heterogeneity in Legion

There are two broad categories of heterogeneity, hardware and software. Hardware heterogeneity has many different facets. The obvious forms of heterogeneity are different processor architectures, instruction sets, data representation formats, and data alignments. These can be accommodated using techniques developed for heterogeneous distributed computing [5][6][7][13][31][32]. The primary question with respect to these techniques is cost: how much overhead do they introduce, and will the overhead cancel out the gains from parallel computation?

A less obvious form of hardware heterogeneity is the configuration of the hosts. Even when two hosts share the same processor architecture, they can differ in ways that can significantly affect both performance and whether a computation can even execute on the hosts. These configuration differences include the clock frequency, the amount of physical memory, the amount of swap and /tmp space, whether the processor has a floating point unit, and so on. These can influence the quality of a scheduling decision.

Another form of host hardware heterogeneity is the model of computation supported by the hardware, e.g., sequential, SIMD, vector, shared memory MIMD (both UMA and NUMA), and distributed memory MIMD. To *exploit* heterogeneity requires that we recognize that different architectures (we include in the notion of architecture the interconnect) are better suited to some problems (or sub-problems) than are other architectures. For example, vectorizable codes run very well on vector super-computers, while scalar codes do not exploit their capabilities. Data-parallel codes such as image processing run very well on SIMD machines, while programs with multiple threads of control do not. These comparative advantages should be exploited.

The management of software heterogeneity presents a challenge in the metasystem environment as well. Even in a homogeneous hardware environment there may be software heterogeneity. Software heterogeneity includes differences in the host operating system (the services and interfaces provided), the process sup-

[4] By "in parallel" we mean that two or more applications may execute concurrently. The applications may be sequential.

port, interprocess communication, the compilers available (the languages and versions of languages available, as well as the quality of the generated code), the file system, and the database systems available. These differences must be masked.

Our solution is presented in detail in [17]. Here we briefly comment on one form of heterogeneity, data representation and alignment. The problem of differing data representations and alignments is familiar, different architectures may represent the same information in different ways. The best known case of this is the classic big-endian versus little-endian problem. Fortunately there has been extensive experience with this problem in the heterogeneous distributed computing community, and there are several well known solutions [22][31][32]. One technique is to convert all data into an intermediate format before it is sent, e.g., XDR [33]. The primary disadvantage with this approach is that all data is coerced twice, adding to the overhead costs.

In Legion data coercion is handled by type-specific coercion functions that transform data from one representation to another. Here the structure of the Legion execution model simplifies the problem. Recall that objects communicate via methods, and the types of the operands are known at compile time. Therefore the IDL compilers can generate code to coerce between representations. The Legion run-time system can then invoke these functions when necessary, i.e., when the sender and receiver use different representations. If they use the same representation, then no coercion is necessary, and no overhead penalty is paid.

3.4 Multilanguage support/interoperability.

While we are committed to the object-oriented paradigm we recognize that Legion will need to support applications written in a variety of languages in order to support existing legacy codes, permit organizations to use familiar languages (C, ADA, Fortran), and support other parallel processing languages. We intend to provide multilanguage support, and interoperability between user objects written in different languages in three ways, by generating object "wrappers" for codes written in languages such as Fortran, ADA, and C; by exporting the Legion run-time system interface and retargeting existing compilers (Figure 1: & Figure 2:) and by a combination of the two.

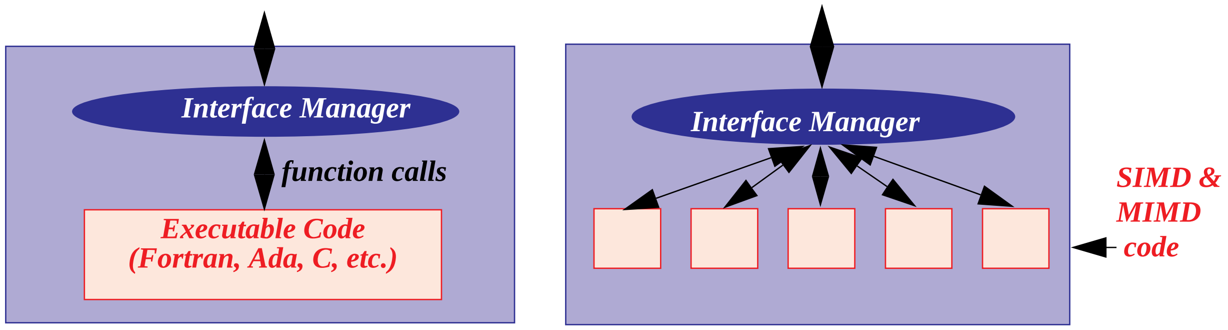

Figure 1: Object wrappers.

Legacy codes and other language codes can be incorporated into Legion applications by encapsulating them in an object wrapper as shown in Figure 1:. Object wrappers can be either hand generated using the Mentat programming language or by using an IDL and an interface compiler. In the IDL case the programmer provides a “class definition” of the object that lists the functions, the type and number of parameters, whether the object has persistent state, etc., and a compiler generates the interface. This is a common technique and is used in the OMG ORB [30].

In addition to encapsulating sequential codes, object wrappers can be used to encapsulate parallel programs as components, such as a C* program for the TMC CM-2 or CM-5. This permits the use of optimized parallel applications as components in larger meta-applications, such as multi-disciplinary optimization (MDO) problems.

Our second method of supporting other languages is to export the Legion interface to compiler writers (Figure 2:). There are a number of parallel languages and parallel processing system efforts underway. Examples include HPF Fortran, Fortran D, Vienna Fortran, pC++, dataparallel C, and others. Each of these projects typically consists of a compiler and a run-time environment. The RTE is often built upon a primitive set of operations, e.g., load, send, receive, broadcast, global sum, etc.,. These operations are either provided by the host operating system, or by a portable communication fabric such as PVM or Express^{TM}.

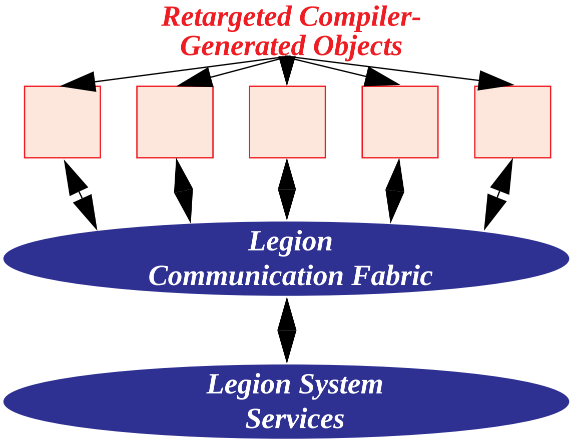

Figure 2: Exposing the Legion run-time system to other parallelizing compilers.

We intend to support these other parallel processing systems by providing the same underlying communication fabric that is typically found. The compiler writers may either retarget their compilers to directly use our communication facilities, or to use a compatibility library that can be constructed. For example, a PVM compatibility library could be constructed that emulates PVM services using Legion communication and system services. We understand that this may result in slightly reduced performance.

Finally, our third method is to combine object wrappers and retargeting compilers for other parallel languages, permitting the use of parallel applications developed under other systems as components of meta-applications. Thus, the component’s components would be distributed across Legion, as opposed to across a single MPP.

3.5 Collaboration support/the national collaboratory

Perhaps the greatest leverage in the NII is to support *people*, to amplify their effectiveness in various ways – and one of the most important of those is to support collaboration among them.

Technology cannot make people collaborate – but it can prevent them from doing so. While electronic mail and other relatively primitive communication have enabled some collaboration among remote researchers, it is still difficult to share instrumentation, simultaneously author a paper, view a common "whiteboard", or do electronically many of the other things that we do so easily when we are collocated.

Legion is *not* a collaboration support tool, but we believe that it is a better base on which to build such tools than is currently available. Current collaboration tools are "point solutions", that is, they each define their own communication protocols, their own storage structures, etc. Legion provides a naturally distributed environment, with location transparency, a common name space and communication/synchronization support. These are just the properties needed for collaboration support tools.

3.6 Resource management

Legion will have a wide variety of resources. Resource classes include CPU cycles, disk and memory space, files, databases, communications channels and bandwidth. For many resources such as CPU's, resource management is a scheduling problem. Scheduling policies and mechanisms must be developed for Legion that accommodate the heterogeneous and distributed nature of the machine. Because the general scheduling problem is NP-hard, the schedulers will be heuristic, and optimal schedules will not be realized.

The objective of scheduling in Legion is to obtain reduced completion time. This is done is two ways: (1) exploit the amount of available CPU resources to increase opportunities for parallel execution and (2) exploit resource affinities within programs to choose the best machine for program *components*. Programs suitable for parallel execution in Legion may contain functional parallel and data parallel components. Scheduling functional parallel components provides an opportunity for exploiting resource affinities or satisfying implementation dependencies. Scheduling data parallel components also provides an opportunity for exploiting resource affinities such as communication topology affinity (e.g., a *1-D* or *2-D* application communication topology is well-suited to a mesh or torus), but the amount of available computational resources provides an even greater opportunity for data parallel components. Scheduling data parallel components is done to exploit communication affinities and the available computational resources.

Scheduling data parallel components in Legion is accomplished in three phases: *processor selection*, *load selection*, and *placement*. Processor selection determines a set of *candidate* processors from the available processors in Legion. Load selection (1) chooses the number and type of processors to use from the candidate set and (2) decomposes the data domain across the chosen processors. Placement maps tasks to processors such that communication time is reduced. Scheduling is performed at runtime when the available processors are known to the system. The scheduling method is a form of *coscheduling* in that all selected processors are scheduled collectively. The method is also *static* in the sense that once a scheduling decision is made it is not changed.

Completion time is determined by two factors: load balance and computation granularity. Computation granularity restricts the amount of parallelism that can be efficiently exploited. This is captured by (1). Load balance ensures that all processors will finish at the same time, a necessary condition for reduced completion time. The data domain is decomposed to achieve load balance. This is captured by (2).

An efficient method for automatically scheduling data parallel components across a LAN (i.e., department-wide Legion) has been developed. The method is a heuristic that chooses the best set of available processors and a decomposition of the data domain that produces the smallest completion time. The method uses information about the program components and the available resources, and a set of communication cost functions to make effective runtime scheduling decisions.

3.7 File/data access

File and data access is one of the most crucial issues for Legion, particularly with respect to providing a seamless environment. Today distributed file systems such as NFS, Andrew, and Locus are commonplace in local area networks. The unified level of service and the naming scheme that they present to their users make them one of the most successful components of contemporary distributed systems. In Legion we intend to provide the same level of naming and access transparency provided in local area networks. This cannot be accomplished though either by directly extending current systems onto a national scale, or by imposing a single file system for both local and Legion access. Instead we propose to adopt a federated file system approach in much the same manner that federated database systems are constructed. The Legion file system will provide naming, access, location, fault, and replication transparency. It will permit users (or library writers) to extend the basic services provided by the file system in a clean and consistent fashion via class derivation and file-object instantiation and manipulation. The extensions that we intend to design and implement ourselves include application specific file objects designed to improve application performance by reducing observed I/O latency.

There is a rich literature in distributed file systems [25]. Issues such as naming, location transparency, fault transparency, replication transparency, and migration have been addressed both in the literature and in one or more existing operational systems. Rather than duplicate those efforts we would rather build on them and extend them into a larger context. The difficulty that arises when borrowing from an existing system is that most of the systems do not have a scalable system architecture, flexible semantics, or they require the imposition of a unified file system model - contrary to our federated file system goal. Therefore we will borrow ideas from the literature, but not implementation, looking more to combine the work of others with our own.

Although we will continue to refer to the Legion “file system”, we intend to create a persistent object space as has been proposed for distributed object management systems [30]. There are several other efforts in the distributed object literature with which we share many goals, for example SHORE [11] and CORBA [27][30]. However, Legion is distinguish from these efforts by the emphasis we place on performance -- Legion expects to provide a high performance computing environment and this goal is paramount. To this end we will focus more on file system support rather than database support.

The model that we will employ is simple and driven by the observation that the traditional distinction between files and other objects is somewhat of an anachronism. Files really are objects - they happen to live on disk, and as a consequence are slower to access and persist when the computer is turned off. We define a file-object as a typed object with an interface. The interface defines the operations that can be performed on it such as the traditional open, seek, and read, as well as other operations defined on a class by class basis such as read_vector, select_sub_space, etc.The interface can also define object properties such as its persistence, fault, synchronization, and performance characteristics. Thus, not all files need be the same, eliminating the need to, for example, provide Unix synchronization semantics, for all files even when many applications simply do not require those semantics. Instead, the right semantics along many dimensions can be selected on a file by file basis, and potentially changed at run-time.

Preliminary work on ELFS, an ExtensibLe File System, has already begun. ELFS will provide language and system support for a new set of file abstractions providing the following capabilities: (1) User specification of caching and prefetch strategies. This feature allows the user to exploit application domain knowledge about access patterns and locality to tailor the caching and prefetch strategies to the application. (2) Asynchronous I/O. ELFS permits overlapping I/O operations, including prefetching, with the applications's computation. (3) Multiple outstanding I/O requests. ELFS allows the application to request data long before it is actually needed. The application can then do some computation while I/O is being processed. By the time the data is needed, it can be had with almost zero latency. (4) Data format heterogeneity. ELFS classes may be constructed so as to hide data format heterogeneity, automatically translating data as it is read or written.

These new file abstractions are structured as a user-extensible class hierarchy with inheritance, thus providing the benefits of object-oriented programming to the application designer. The programmer can then extend basic file abstractions with type-specific operations.

To date our efforts with ELFS have been confined to high-performance, application specific file objects. We are currently extending our efforts in support of the campus-wide virtual computer to include the capability to transparently access files that “live” on another file system, i.e., the file is not visible (has not been cross mounted) and cannot be accessed using the native file system. Thus we will construct a simple federated file system composed of multiple file systems without cross-mounting.

3.8 Fault-tolerance

In a system as large as Legion, it is certain that at any given instant several hosts, communication links, and disks will have failed. Thus, dealing with failure and dynamic reconfiguration is a necessity. Current parallel processing systems, including Mentat, do not address fault-tolerance. As is the case in the management of heterogeneity there is an extensive literature in reliable distributed computing systems. We consider two different fault tolerance problems, fault tolerance of Legion itself, and application fault-tolerance.

There is a trade-off between performance and different levels of fault tolerance. It is not necessary or desirable to be able to withstand all forms of faults and continue execution as if nothing had happened. The limit of our concern is with fail-stop faults of hardware components, including both processors and the network. Therefore, we will not address issues such as software faults or Byzantine failure.

Legion system components must be fault tolerant to the extent necessary and possible without compromising our high-performance objectives. For example, the underlying message system will guarantee the delivery of messages in the presence of lost, reordered, or duplicate packets, or less than complete network failure. Further, if a host should fail, then Legion will reconfigure itself to remove that host from the current configuration, and will reconfigure again to include the host when the host recovers. Similarly, Legion will continue operation under network partition by treating the non-accessible hosts as dead. The difficulties often associated with partition merge, e.g., bringing divergent copies of a database to a consistent state, will be avoided in Legion by ensuring that all Legion system owned databases do not require a globally consistent state, and by providing mechanisms for the lazy propagation of updates. Further, Legion will not define the semantics of operation under partition and merge of user defined persistent objects.

Our philosophy on application fault tolerance is that an application should not pay for fault-tolerance it does not need. Therefore, applications will be able to tune their fault tolerance requirements by specifying a level of fault tolerance, and a penalty they are willing to pay in terms of recovery time.

Briefly our implementation will exploit the properties of objects and their realization in the run-time. Recall that there are stateless and stateful objects. They are treated differently. Fault-tolerance for stateless objects can be provided by noting when a method begins execution and by treating the message system in a database-like fashion. When the invocation of a method begins the messages are read from the database without consuming them. Then, when the method completes, the arguments are consumed, and the results placed into the database in one atomic action. If the method fails during execution, either due to an internal failure, or due to hardware failure, the system reacts accordingly. In the case of an internal failure, e.g., memory fault, the user is notified and the arguments that caused the failure are available for use with a debugger. If a host has failed, then the method is restarted on another host using the same arguments.

Stateful objects present a more difficult problem and the search for the best way to specify and implement differing levels of fault-tolerance will be a research priority. The difficulty lies not with providing fault tolerance for a single object, that can be accomplished with check-pointing and transactions on the message system "database". The difficulty lies with the interaction of multiple objects and guaranteeing that they see a consistent view of the world.

3.9 Protection/security/authentication

The issue of security in general is becoming more important as computers assume more safety critical and economically sensitive tasks, and as the NII facilitates interoperation of systems. Legion cannot solve the security problem, but a reasonable level of privacy and integrity of data must be provided or the system will not be used.

Legion will run on top of whatever operating system is available on the participating host, thus at first blush it may seem that the issue of security would be determined by the lowest common denominator of those host systems.

The *first rule* of Legion security, like the Hippocratic Oath, will be to "do no harm". In the first instance, that means that there should be no possibility of host data being compromised by a Legion task. In the second instance it means that no Legion user's data should be compromised by a rogue host. The mechanisms are reasonably well understood and include:

- execution of Legion tasks with “least privilege” on the host,
- storing no persistent objects on a “foreign host”,
- using cryptographic authentication protocols to verify that a server is valid and that a service request originated with a valid Legion task,
- digital signatures to validate that data has not been tampered with.

In addition, upon request of either the host or Legion user for greater security, more costly techniques can be employed – for example:

- messages can be encrypted,
- scheduling can be restricted to certain classes of nodes,
- white noise can be injected into the communication stream.

The second rule of Legion security will be to provide a better model and mechanism for Legion-to-Legion security than is provided by the underlying hosts. The object-oriented computational model naturally suggests a capability-based access control model in which the “rights” are type (class) specific and map one-to-one to the methods of the class. The distributed and heterogenous nature of the system suggests the use of encrypted capabilities to ensure that they cannot be forged.

3.10 High-performance Communication Protocols

Communication among geographically distributed (and perhaps architecturally distinct) computers is accomplished by using a common suite of communications protocols. From the point of view of the application program, the most important of these are the protocols which implement the *network* and *transport* layers of the OSI reference model. The transport protocol traditionally provides end-to-end, reliable, sequential, non-duplicated delivery of data across an arbitrary subnet, while the network protocol accomplishes routing, route determination, fragmentation, and packet lifetime control. Two traditional protocols for these two tasks are the *Transmission Control Protocol (TCP)* and the *Internet Protocol (IP)*. While TCP and IP are often considered to be a single unit, they are not; in fact, there are many transport protocols that run over IP and provide different types of end-to-end services.

For Legion, we propose to use the *Xpress Transfer Protocol (XTP)* as our transport protocol. XTP operates over IP (as well as other network protocols, such as CLNP) and thus is usable in any computing device currently attached to the Internet. We have chosen XTP instead of TCP because XTP provides all the functionality of TCP, plus additional services especially designed for high-throughput, low-latency communications. Examples of these additional services include: reliable datagrams (one-time messages that nevertheless must be successfully delivered); transactions (request/response queries); rich message priority mechanism (transmitters, receivers, and all interior routers are always operating on their most important messages); selective retransmission (when packets are lost, only the missing ones are retransmitted, unlike the go-back-n mechanism in TCP); multicast (every connection can be a multi-peer connection); and multicast group management (reliability is controlled by the application, and can vary from none to complete).

In addition to file transfers and process-to-process communication, XTP also supports multimedia applications. For instance, using a combination of XTP’s modes, we can establish distribution of a synchronized audio/video bitstream from a single transmitter to any number of receivers. For workstations equipped the audio/video hardware, Legion can thus support a teleconferencing utility among cooperating end stations.

While XTP has been implemented for various local area networks (e.g., Ethernet, token ring, FDDI) and for various operating systems (e.g., Unix, DOS, Windows, and some real-time kernels), global interoperability requires specifically that we operate over both the existing Internet and the burgeoning National Information Infrastructure. While the former has been largely accomplished, we are currently porting XTP to operate over Asynchronous Transfer Mode (ATM) host adapters and ATM switches in anticipation of the latter. By supporting three protocol combinations underneath XTP (IP alone, ATM alone, and IP over ATM), we will be able to support process-to-process messaging among applications located anywhere in the world.

4.0 Summary and Project Status

Our agenda consists of three stages, (1) the construction of a campus-wide virtual computer at the University of Virginia, (2) packaging the campus-wide system for preliminary experimentation and use by Legionnaires, and (3) expansion to a nationwide demonstration system. Each of these three stages will build upon the previous.

Before any major project is undertaken, one must ask how to measure success. In parallel processing success is measured by application performance (speedup, MFLOPS) and the flexibility and ease of use of the tool. These are important metrics for Legion as well, but they are not the only metrics. Other important metrics include acceptance by the user community, fault-tolerance, cost per *used* MIP/FLOP, and whether tasks can be performed that were not possible before (e.g., run an application in Virginia on data that resides at JPL and NASA-Langley, or collect and use data in real time from sensors in orbit, but have it look like any other “file”).

Application performance will be measured for a variety of real-world applications, as well as selected kernel codes and parallel processing benchmarks. The applications will be drawn from a diverse set of disciplines: biology, physics, electrical engineering, chemistry, economics, radio astronomy, and command and control. The applications will possess different granularity characteristics, as well as different amounts of latency tolerance. It is not our intent, however, to show that all applications will be capable of exploiting the nationwide resources of Legion. Some applications, in particular those with inherently small granularity, or that are latency intolerant, will remain best suited to local operation, e.g., on a single processor or on a single tightly-coupled parallel processor.

4.1 Construction of a campus-wide virtual computer (CWVC)

The campus-wide virtual computer is a direct extension of Mentat to a larger scale, and is a prototype for the nationwide system. Even though the CWVC is much smaller, and the components much closer together, than in the envisioned nationwide Legion, it still presents many of the same challenges. The processors are heterogeneous, the interconnection network is irregular, with orders of magnitude differences in bandwidth and latency, and the machines are currently in use for on-site applications that must not be negatively impacted. Each department operates essentially as an island of service, with its own NFS mount structure.

Table 1-a. Resources

Computer Type	Quantity
SPARC IPC	38
SPARC 1+	1
SPARC2	13
SPARC10	2
SGI Indigo	6

Table 1-b. Complib - 42,864 sequence target library

Number of workers	Best Time (sec)	Average Time (sec)	Best Total Time (sec)	Average Total Time (sec)
sequential (IPC)			10,876	
10	567	733	595	769
20	841	874	892	927
30	636	759	693	828
40	441	467	544	546
50	398	443	481	509
60	343	411	443	450
70	323	332	376	387

The CWVC is both a prototype and a demonstration project. The objectives are to demonstrate the usefulness of network-based, heterogeneous, parallel processing to university computational science problems; provide a shared high-performance resource for university researchers; provide a given level of service (as measured by turn-around time) at reduced cost; and act as a testbed for the nationwide Legion.

The prototype consists of over sixty workstations and is now operational. In [14] we present the performance of two production applications that we have used to test the efficacy of our approach: *complib*, a biochemistry application that compares DNA and protein sequences, and ATPG, an electrical engineering application that generates test patterns for VLSI circuits. The performance results are encouraging. Table 1 presents early performance results for *complib*.

4.2 The nationwide demonstration Project

Once the CWVC has successfully demonstrated the efficacy of our approach on a small scale, we will turn our attention to the nationwide Legion. There are significant differences between this environment and the CWVC: latencies are much larger, there is no shared file system and little chance of forming one, there are more parallel machines, there are more powerful machines, the member organizations are significantly different, accounting is used at many sites, and protection and security issues are increasingly important.

The first step is to identify potential member organizations that would be interested in participating in the demonstration project. We have identified several sites to approach with which we have existing working relationships and which have agreed to participate.

Once a sufficient number of sites have signed on, the next step is to attend to the interconnection network. To be successful on any but the most trivially parallel applications, a high-bandwidth network between the sites and high-performance protocols for the network are necessary. This is a key aspect and is already under investigation.

Once the sites and the interconnection network are on-line, application performance testing can begin. The objective of the testing is to measure application performance and the impact on local computing, experiment with different scheduling algorithms, and to demonstrate that a nationwide virtual computer can out-perform the resources available at any single site.

5.0 References

[1] S. Ahuja, N. Carriero, and D. Gelernter, “Linda and Friends,” *IEEE Computer*, pp. 26-34, August, 1986.

[2] H. Bal, J. Steiner, and A. Tanenbaum, “Programming Languages for Distributed Computing Systems,” *ACM Computing Surveys*, pp. 261-322, vol. 21, no. 3, Sept. 1989.

[3] J. Boyle et al., *Portable Programs for Parallel Processors*, Holt, Rinehart and Winston, New York, 1987.

[4] A. Beguelin et al.,‘‘A Users’ Guide to PVM (Parallel Virtual Machine)’’, Oak Ridge National Laboratory TM-11826.

[5] G. Bernard et al.,, “Primitives for Distributed Computing in a Heterogeneous Local Area Network Environment”, *IEEE Trans on Soft. Eng*. vol. 15, no. 12, December 89.

[6] B. N. Bershad, and H. M. Levy, “Remote Computation in a Heterogeneous Environment.” Tech. Rep. 87-06-04, Dept. of Computer Science, University of Washington, Seattle, June, 1987.

[7] B. N. Bershad, et al., “A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems,” *IEEE Trans. Software. Eng. SE*, vol. 13, no. 8, pp. 880-894, August, 1987.

[8] F. Bodin, et. al., “Distributed pC++: Basic Ideas for an Object Parallel Language,” *Proceedings Object-Oriented Numerics Conference*, pp. 1-24,Sunriver, Oregon, April 25-27, 1993.

[9] D. Callahan and K. Kennedy,”Compiling Programs for Distributed-Memory Multiprocessors” *The Journal of Supercomputing*, no. 2, pp. 151-169, 1988, Kluwer Academic Publishers.

[10] N. Carriero and D. Gelernter, “Linda in Context,” *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April, 1989.

[11] M.J. Carey, et. al., “Shoring Up Persistent Applications,” *to appear, SIGMOD 1994*.

[12] A.L.Cheung, and A.P. Reeves, “High Performance Computing on a Cluster of Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 152-160, Syracuse, NY, Sept., 1992.

[13] P. B. Gibbond, “A Stub Generator for Multi-Language RPC in Heterogeneous Environments,” *IEEE Trans. Software. Eng. SE*, vol. 13 , no. 1, pp. 77-87, January, 1987.

[14] A. S. Grimshaw, D. Shiflet, and A. Nguyen-Tuong, “Campus-Wide Computing: Early Results Using Legion at the University of Virginia,” *submitted to Supercomputing ‘94*.

[15] A. S. Grimshw, J. B. Weissman, and W. T. Strayer, ‘‘Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,’’ submitted to *ACM Transactions on Computer Systems*, July, 1993.

[16] A. S. Grimshaw, E. A. West, and W.R. Pearson, “No Pain and Gain! - Experiences with Mentat on Biological Application,” *Concurrency: Practice & Experience*, pp. 309-328, Vol. 5, issue 4, June, 1993.

[17] A.S. Grimshaw, J.B.Weissman, E.A. West, and E. Loyot, “Meta Systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems,” *to appear Journal of Parallel and Distributed Computing*.

[18] A. S. Grimshaw, W. T. Strayer, and P. Narayan, “Dynamic Object-Oriented Parallel Processing,” *IEEE Parallel & Distributed Technology: Systems & Applications*, pp. 33-47, May, 1993.

[19] A. S. Grimshaw, “Easy to Use Object-Oriented Parallel Programming with Mentat,” *IEEE Computer*, pp. 39-51, May, 1993.

[20] A. S. Grimshaw, and E. Loyot Jr., ELFS: Object-Oriented Extensible File Systems, University of Virginia, Computer Science TR 91-14, 1991.

[21] P.J. Hatcher, “A Production-Quality C* Compiler for Hypercube Multicomputers,” *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Williamsburg, VA, April 21-24, 1991.

[22] M. Jones, R. F. Rashid, and M. R. Thompson, “An Interface Specification Language for Distributed Processing.” *Proceedings of the 12th ACM Symposium on Principles of Programming Languag*es, pp. 225-235, 1985.

[23] J.A. Kaplan and M.L. Nelson,”A Comparison of Queueing, Cluster, and Distributed Computing Systems,” NASA Technical Memorandum 109025, NASA LaRC, October, 1993.

[24] Ashfaq Khokhar, et. al., “Heterogeneous Supercomputing: Problems and Issues,” *Proceedings of WHP 92 Workshop on Heterogeneous Processing*, IEEE Press, pp. 3-12, Beverly Hills, CA, March, 1992.

[25] E. Levy, and A. Silberschatz, “Distributed File Systems: Concepts and Examples,” ACM Computing Surveys, vol. 22, No. 4, pp. 321-374, December, 1090.

[26] D. B. Loveman, “High Performance Fortran,”*IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 25-42, February, 1993.

[27] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie, “Distributed Object Management,” *International Journal of Intelligent and Cooperative Information Systems*, vol. 1, no. 1, June 1992.

[28] J.H. Morris, et al., ‘Andrew: A distributed personal computing environment’, *Communications of the ACM*, vol. 29, no. 3, March 1986.

[29] N. Nedeljkovic, and M.J. Quinn, “Data-Parallel Programming on a Network of Heterogeneous Workstations,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 28-36, Syracuse, NY, Sept., 1992.

[30] J.R. Nicol, C.T. Wilkes, and F.A. Manola, “Object-Orientation in Heterogeneous Distributed Systems”, *IEEE Computer*, vol.26, no. 6., pp. 57-67, June, 1993.

[31] D. Notkin, N., et al.,“Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity,” *Communications of the ACM*,vol. 30, no. 2, pp. 132-140, February, 1987.

[32] D. Notkin, et al., “Interconnecting Heterogeneous Computer Systems,” *Communications of the ACM*, vol. 31, no. 3, pp. 258-273, March, 1988.

[33] Sun Microsystems. *External Data Representation Reference Manual.* Sun Microsystems, Jan. 1985.

[34] V.S. Sunderam, “PVM: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

[35] B. Walker, et al., “The LOCUS Distributed Operating System,” *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, N. H., Oct.) ACM, New York, 1983.

[36] Min-You Wu, and G.C. Fox, “A Test Suite Approach for Fortran90D Compilers on MIMD Distributed Memory Parallel Computers,” *Proceedings of the First Symposium on High-Performance Distributed Computing*, pp. 393-400, Syracuse, NY, Sept., 1992.