

Campus-Wide Computing: Early Results Using Legion at the University of Virginia¹

Andrew S. Grimshaw, Anh Nguyen-Tuong, Mike Lewis, and Mark Hyett

{grimshaw | nguyen | mlewis | mrh2e} @ virginia.edu

Department of Computer Science

University of Virginia

Abstract

The Legion project at the University of Virginia is an architecture for designing and building system services that provide the illusion of a single virtual machine to users, a virtual machine that provides both improved response time via parallel execution and greater throughput. Legion targets workstation clusters and larger wide area assemblies of workstations, supercomputers, and parallel supercomputers. We have built a working Legion prototype, called the Campus-Wide Virtual Computer (CWVC). The CWVC extends an existing object-oriented parallel processing system by aggressively incorporating lessons learned in the last twenty years of heterogeneous distributed computing. In this paper, we describe the challenges that we overcame to realize a working CWVC, and we characterize the performance of a production biochemistry application.

1. Introduction

Computationally demanding applications challenge organizations to provide powerful computing resources. Traditionally, expensive supercomputers have often been used to satisfy the applications' demand for computing cycles. An alternative less costly solution has emerged. Organizations are attempting to harness the power of networks of high-performance workstations and personal computers, an approach that can take advantage of existing underutilized resources, and costs less per megaflop than a supercomputer. However, without system software to tie the machines together, the complexity of the environment overwhelms most users, who typically cannot exploit its full power. To become a unified computing entity, a cluster of workstations requires software to manage the resources.

The Legion project² at the University of Virginia is an architecture for designing and building system services that provide the illusion of a single virtual machine to users, a virtual machine that provides both improved response time via parallel execution and greater throughput [8]. Legion targets workstation clusters and larger wide area assemblies of workstations, supercomputers, and parallel supercomputers. Legion tackles problems not solved by existing workstation based parallel processing tools; the system will enable fault-tolerance, wide area parallel processing, inter-operability, heterogeneity, a single global name space, protection, security, efficient scheduling, and comprehensive resource management.

Instead of constructing Legion from scratch, we have chosen an evolutionary approach. We began by constructing a Campus-Wide Virtual Computer (CWVC) testbed based on Mentat [7], a robust object-oriented parallel processing system that supports execution across heterogeneous platforms. The strategy of enhancing an existing system can help eliminate the long lead times and the uncertainty associated with starting from scratch. Further, applications that run on the prototype will run in the eventual Legion system, enabling ideas and system implementations to be tested using real applications.

In many ways, a university campus computing environment is a microcosm of a wide area network. Certainly, the CWVC is much smaller than the envisioned worldwide Legion, and its components are physically much closer to one another. But the campus environment does present many of the same

¹. This work is partially funded by NSF grants ASC-9201822 and CDA-8922545-01, National Laboratory of Medicine grant (LM04969), NRaD contract N00014-94-1-0882, and ARPA grant J-FBI-93-116.

². For more information about Legion, including technical reports, see <http://www.cs.virginia.edu/~legion>.

challenges. The computational resources at a university are owned and operated by many different departments, a single shared name space does not exist, and departments rarely share resources. The processors are heterogeneous and fail often, the interconnection network is irregular, with orders of magnitude differences in bandwidth and latency, and the machines are currently used for on-site applications whose performance must not degrade due to our system. Further, each department operates essentially as an island of service with its own NFS mount structure, and trusts only machines on its own island.

1.1. Challenges

The prototype currently consists of over one hundred workstations and an IBM SP-2 in six buildings. Before the system could become useful to users, we had to overcome several challenges; simply scaling Mentat to run on more machines would not suffice. The first challenge was to construct a federated file system and unified name space so that files –both executables and user data– could be accessed from any host in the system. The second challenge was to enable the system to deal gracefully with host and network failure. Hosts in a university environment fail regularly and when they do, the system must continue to operate without interruption. The third challenge was to provide effective tools for debugging parallel applications in the campus-wide computing environment. The fourth challenge was support for interoperability and other programming models as no single paradigm can fit the needs of all users. We currently support the Mentat Programming Language, an extended C++ with compiler support for concurrency, and PVM, a *de facto* standard in the parallel processing community.

Two final challenges are rooted in human nature rather than technical necessity. The fifth challenge was to provide autonomy to workstation owners and system administrators. Users and local system administrators should be allowed to control resource utilization on their hosts in order to enforce local scheduling policies and to throttle utilization. Without the ability to effect limit utilization, users would be reluctant to add their resources to the pool. We call this challenge “pain management.” The sixth and final challenge discussed here was resource accounting. To avoid the tragedy of the commons, where everyone uses resources but no one contributes them, the system must track how much resource is contributed (offered and used) and consumed by each user. Users who contribute more than they use can be rewarded, and those who consume more than they contribute can be charged. How users are rewarded and charged is a policy issue that we have not yet addressed. Keeping track of resource utilization requires mechanism, which we built into the CWVC. In solving the above challenges we had to ensure that system performance would not be degraded – for many users, better performance is the primary benefit of the system.

1.2. Roadmap

In this paper, we present a brief background of the Mentat and Legion projects, followed by a description of our early results using the CWVC. We outline the shape of our solutions to the six challenges described above, and then characterize system performance using a biochemistry application, called “complib,” which compares libraries of DNA and protein sequences. The performance results are encouraging. Several other production applications have been developed by and for university researchers, including planetary atmosphere simulations, steric acid circulation tank simulations, solid state circuit simulations, parallel genetic algorithms, and 3D image segmentation.

2. Background

2.1. Mentat

Mentat³ is a high performance, object-oriented parallel processing system. There are two primary aspects of Mentat: the Mentat Programming Language (MPL) and the Mentat run-time system [9]. MPL is

³. Information on Mentat is available on the WWW at <http://www.cs.virginia.edu/~mentat> and in [7][22].

an object-oriented programming language based on C++. The basic idea in MPL is to allow the programmer to specify those C++ classes that are of sufficient computational complexity to warrant parallel execution. This is accomplished using the `mentat` keyword in the class definition. Instances of Mentat classes are called Mentat objects; each constitutes a separate address space. Programmers use instances of Mentat classes much as they would any other C++ class instance. The Mentat run-time system and compiler cooperate to automatically detect and manage the data and control dependencies between Mentat class instances involved in invocation, communication, and synchronization. The basic philosophy is that programmers know their application best and should be the ones making decomposition decisions while deferring the error-prone tasks of managing communication and synchronization to the compiler and run-time system.

2.2. Legion

Legion is a metaseystems software project at the University of Virginia. Our goal is a highly usable, efficient and scalable system based on solid principles. We have been guided by our own work in object-oriented parallel processing, distributed computing, and security, as well as by decades of research in distributed computing systems. When complete, Legion will provide a single, coherent virtual machine that addresses scalability, programming ease, fault tolerance, security, site autonomy, etc. In short, we believe Legion is a conceptual base for the sort of metaseystem we seek.

We envision Legion consisting of millions of hosts and trillions of objects co-existing in a loose confederation tied together with high-speed links. The user will have the illusion of a very powerful computer on her desk. She will sit at her terminal and manipulate objects. “Terminals” include workstations, immersive environments such as head-mounted displays or the CAVETM, and portable Personal Digital Assistants. The objects she manipulates will represent data resources such as digital libraries or video streams, applications such as teleconferencing or physical simulations, and physical devices such as cameras, telescopes, and linear accelerators. Naturally the objects being manipulated may be shared with other users, allowing the construction of shared virtual workspaces.

It is Legion’s responsibility to support the abstraction presented to the user, to transparently schedule application components on processors, manage data migration, caching, transfer, and coercion, detect and manage faults, and ensure that the user’s data and physical resources are adequately protected.

Before we can realize the Legion vision, we must overcome several technical software challenges. Other researchers are addressing the hardware challenges; the solutions will become the enabling technologies that provide the opportunity for Legion. The software challenges revolve around ten central objectives: site autonomy, an extensible core, a scalable architecture, an easy-to-use seamless computational environment, high performance via parallelism, a single global name space, security for both users and resource providers, management and exploitation of resource heterogeneity, multi-language support and interoperability, and fault-tolerance. We examine these issues in more detail in [10]. In addition to the purely technical issues, political, sociological, and economic issues further complicate the purely technical challenges. These include encouraging the participation of resource-rich centers and discouraging the tendency to free-ride.

The principles of the object-oriented paradigm are the foundation for the construction of Legion. We plan to exploit the paradigm’s encapsulation and inheritance properties, and to benefit from 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 applications and libraries for wide area parallel processing using only low-level message passing services. Such tools require the programmer to address the full complexity of the environment, but the difficult problems of scheduling, load balancing, managing faults, etc., are likely to overwhelm all but the best programmers.

Complementing our use of the object-oriented paradigm is one of our driving philosophical themes—*we cannot design a system that will satisfy every user’s needs*. We must design Legion to allow users and class implementors the greatest flexibility in the semantics of their applications: We must, therefore, resist the temptation to provide “the solution” to a wide range of system functions. Users should be able, whenever possible, to select both the *kind* and the *level* of functionality, and make their own trade-offs between function and cost.

Neither the “kind” nor the “level” of functionality are linearly ordered, but a simplistic model is that of a multi-dimensional space. The needs of users will dictate where they need to be and/or can afford to be in this space; we, the designers of the supporting conceptual system have no way of knowing what those needs are, or what they will evolve to be in the future. Indeed, if we were to dictate a system-wide “solution” to almost any of the issues raised in our list of objectives we would preclude large classes of potential users and uses.

3. The Campus-Wide Virtual Computer

Simply running Mentat on all the machines at the university would not realize “campus-wide computing.” Mentat, like other existing parallel processing systems, does not deal with system faults, non-overlapping file systems, and other problems that make computing in a distributed system more difficult than on an MPP. Further, we do not believe that the Mentat Programming Language is the best language for all applications. Therefore, to be successful, Legion will need to support other models and languages. In this section we present the CWVC solutions to the problems of files and I/O, fault-tolerance, pain management, accounting, debugging parallel applications, and multiple programming models (Table 1).

Problem	Tools Available	Section
Multiple separate file systems	Federated File System	3.1.
Host / network failure	Automatic system reconfiguration and limited application fault-tolerance	3.2.
Multiple resource owners & sharing of resources	Detailed accounting information and “pain management”	3.3., 3.4.
Debugging parallel programs is hard	Post-mortem playback using off-the-shelf debuggers	3.5.
Writing parallel applications	CWVC-aware PVM, Mentat Programming Language, Fortran wrappers	3.6.

Table 1 Campus-Wide Virtual Computer Toolsets

3.1. The Federated File System

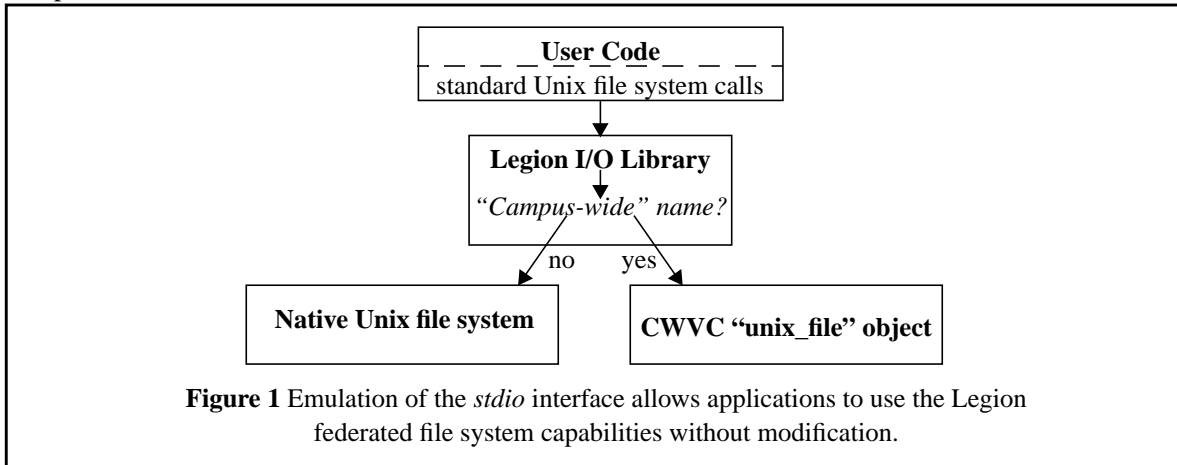
Hosts in different organizations usually do not share a single file system. This presents at least four difficulties, (1) application binaries may not be present at all sites, (2) application components may not be able to read and write files that they require for correct execution, (3) sharing of data and results between collaborators at different sites requires either sending the files by email or copying them via ftp, and (4) remote data files must be copied in their entirety into the local environment before they can be used, resulting in wasteful copies, out-of-date data, and inconvenience.

One solution would be to extend an existing file system such as AFS or NFS to both the campus-wide and worldwide system. Unfortunately, some file systems such as NFS simply will not scale well enough. Further, NFS requires that all users have the same user id on all hosts, a requirement that will not hold in wide area systems. The Andrew file system is scalable. However, we cannot impose a file system standard on participating organizations.

Our solution is to construct a Federated File System using the local host file systems as components. The approach consists of two parts—constructing a unified campus-wide name space on top of the existing file system, and modifying a set of standard I/O libraries that interact with the unified file space and file objects. A file is in the unified name space if its name (complete path name) is found in one of the name servers, otherwise it is a local file. The name servers keep track of the file’s name, e.g., “/legion/grimshaw/mywork1”, and a set of file attributes. The attributes include the class of the file (e.g., `unix_file`), restrictions on the placement of class instances (e.g., on hosts with a “.cs.virginia.edu” suffix), and an initialization string to be passed to an object instance on instantiation. This string is most often the local Unix path name of the actual file data.

We also provide a set of library routines that support the *stdio* interface—*open()*, *close()*, *read()*, *write()*, etc. These library routines are linked into applications and intercept calls to the I/O system. All *open()* calls are first examined to determine whether they are in the unified name space or local files. Operations on local files are passed onto the host operating system. Campus-wide file operations are trapped and passed onto the *unix_file* objects. This technique of trapping file operations is not unique to Legion and was first used in Unix United.

In addition, the Federated File System includes a set of shell commands to manipulate the campus-wide namespace. The operations are analogous to those that manipulate the Unix name space, e.g., *mkdir*, *rm*, *ln*, etc. Files enter the campus-wide namespace via one of three mechanisms, (1) they are created in the unified name space, (2) they are moved into the unified name space, or (3) they are linked into the unified name space (similar to a soft-link).



3.2. System fault-tolerance

In a system as large as the CWVC machines often fail. Mentat, and other network parallel processing systems do not typically address fault-tolerance. To be usable the CWVC should be available when users need it, and should tolerate host, network, and application failure. Contemporary computer users have very high expectations for system availability, thus the CWVC must be resilient to faults or no one will use it.

Before we can describe the current fault-tolerance mechanism, some background is required. Each host in the CWVC runs at least two daemons (other than the fault-tolerance daemons). The instantiation manager (IM) is responsible for scheduling objects, keeping track of all CWVC objects on the host, monitoring load and other management functions. The token matching unit (TMU) supports language features [9] such as stateless objects. A system configuration file names the hosts in the CWVC⁴. The set of IM's running on those hosts define the system. If an IM fails, then the CWVC cannot use that host. Further, user programs, user objects, and IM's communicate with IM's to carry out their functions. If an IM on a host fails (either because of a program fault or host failure), some objects on other hosts may never receive an expected response, and will "hang." To eliminate this form of failure, the CWVC must either ensure that hosts and IM's never fail, which is impossible, or it must deal with failure when it occurs.

To deal with failure, we have constructed a new class of daemons, called "phoenix," that monitor the system for failure. Phoenix is responsible for restarting system components if possible. If a component cannot be restarted, phoenix notifies the remaining IM's of a configuration change. Each IM then notifies all objects on its host. If a host has been removed from the configuration due to failure, then phoenix will begin to monitor the host for recovery. When the host has recovered, phoenix will restart the CWVC on that host and will notify the other IM's of the configuration change.

⁴. This design is non-scalable as we currently require a system-wide configuration database. Future versions of the system will use a different mechanism and allow for a more dynamic configuration.

The phoenix instances are arranged in a tree structure. The *root* phoenix starts *cluster* phoenix instances on a single host in each cluster. These cluster phoenix instances in turn start a *leaf* phoenix instance on each host in the cluster. The leaf phoenix instances monitor the local daemons, restarting them as necessary. The root phoenix monitors the leaf instances. When one fails (usually due to a host failure) the root restarts it. A root phoenix manages a single fault-tolerance administrative domain and represents a single point of failure for that domain. We allow disjoint fault-tolerance domains to be merged to form a single system. In case of a clean network partition, the merged partitions would once again become their own island of services.

Transparent application fault-tolerance is being investigated but is not currently implemented. We believe that fault-tolerance services are a necessary condition for success of any large scale system such as Legion. We already have an experimental version of the system that provides fault-tolerance for stateless objects and applications that use them [19]. This is an important first step as it encompasses a large class of applications.

3.3. Pain management

A workstation owner may feel “pain” when a CWVC job is executing on her machine; that is, her local application’s performance may degrade due to CWVC applications. In response to this problem, we have constructed a CWVC “thermostat” (Figure 2) that permits resource owners to impose maximum resource usage limits (in percent CPU and physical memory) on CWVC programs running on their hosts. The thermostat mechanism is similar to many home thermostats. The owner can select resource availability during different time intervals. Just as with a home thermostat, if the owner is unhappy with the current setting, she may change it at any time. As with a home thermostat, it may take several seconds before the load is adjusted to the new level.

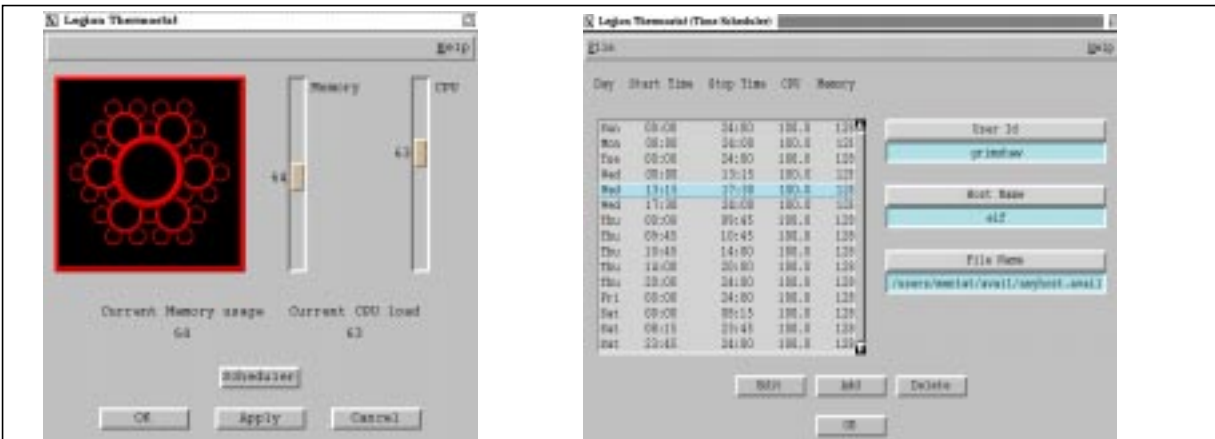


Figure 2 CWVC Thermostat. Authorized users can specify a resource schedule and can modify the current resource limits. The interface on the left controls current resource limits. By changing the sliders, resource limits may be modified temporarily. The “scheduler” button brings up the interface on the right, which allows authorized users to change the daily schedule.

The system guarantees that Legion resource consumption will stay below the limits imposed by the user. The available resources are divided among CWVC user objects running on the host. Thus, some user objects will be “throttled,” potentially resulting in longer execution times for applications. We feel that the trade-off between autonomy and application performance must be made in favor of the local user, otherwise resource owners may withdraw their resources.

To encourage resource owners to participate and to specify high resource limits, the resource owner receives credits when resources are consumed and when resources are offered *even if they are not used*. The amount of credit received depends on the type of system, the time of day, and how much resource was offered and used. Different amounts of credit are received for offered but not used, and offered and used

resource. The scale factors (for type of system and time of day) essentially serve as prices, which are set by policies that are determined by system administrators.

3.4. Accounting

If Legion and the CWVC are successful, then a simple X-Windows interface will allow users to access the set of resources managed by the system. Users will be tempted to use just this “client” interface, and will have little incentive for actually providing resources. Soon no new resources would be provided, resulting in an impoverished system. To avoid this classic “tragedy of the commons,” mechanism and policy are required to encourage good community behavior. For Legion and the CWVC, an accounting system monitors resource contribution and consumption.

Accounting for resource consumption is accomplished by associating each instantiated user object with an owner and a user. The system then monitors user object activity. Resource consumption is monitored on an object-by-object basis. Objects are linked with run-time libraries that collect information such as the amount of CPU time used, the number of messages sent and received, the total volume of data moved into and out of the object, and the number of method invocations performed. When an object terminates it forwards this information to the instantiation manager. The IM then places the resource data into a host specific accounting database. Periodically, the system collects accounting data, merges it into a single database, and generates resource reports.

Resource contribution is monitored similarly. Actual resource contribution is derived from the resource consumption database; if resources on a host were used by an object, then they were “contributed” by that host. “Offered” resources are determined by maintaining thermostat logs. Recall that the thermostat is used to throttle resource consumption on a host by restricting usage to a particular percentage. The thermostat log indicate when thermostat settings were changed on a host.

Both resource contribution and consumption are scaled using the resource scale file. The file includes scales for host types, times of day, and whether the resources were consumed, offered, or contributed. Scales balance differences in resource values; for example, a Sparc IPC CPU second at 1:00 AM is not worth nearly as much as an SGI CPU second at noon. Using the accounting files and the resource scale file, we can calculate a resource balance for each user. The resource balance is the user’s amount of surplus or deficit. A system-wide surplus, available to system administrators, can be generated by maintaining a spread between the price for resources consumed and resources offered.

The accounting mechanism indicates who is using and contributing resources. Without a policy on resource consumption, collecting the information is an exercise in programming only. Policy is still being worked out with resource holders. We envision a system that allows users with chronic resource deficits to either buy (using dollars) more resource or receive “grants” from the system. The grants will come out of the system-wide surplus. The funds generated can be used to pay users with chronic surpluses, or can be re-invested in additional equipment.

3.5. Debugger

A common problem in using a parallel processing environment is the lack of good debugging tools. Even when tools are present they are often difficult to use and require programmers to learn another debugging environment. The end result is that the most common technique for debugging parallel applications has been the insertion of print statements. Unfortunately, this technique quickly breaks down when debugging programs consisting of hundreds or thousands of objects. Our philosophy in designing debugging tools for Mentat was that users should not have to learn a new environment to debug their programs, especially when they have already invested time and effort in learning a sequential debugger.

The Mentat Assistant Debugger (MAD) is a set of tools that enables programmers to debug their Mentat/Legion applications with the debugger of their choice. MAD supports a style of debugging known as post-mortem debugging: debugging occurs after a Mentat program runs to completion or until an error occurs. MAD is based on the record and replay technique and consists of two phases. In the recording phase, objects that comprise a Mentat application log their incoming messages to a file. In the playback phase, objects can

faithfully reproduce their original execution by extracting the appropriate messages from the log file. The programmer can then replay a specific object, i.e. reproduce its execution, under the control of a sequential debugger and use traditional debugging techniques. MAD is not designed to handle timing dependent errors (Heisenbugs) and is best suited for repeatable errors which usually manifest themselves by a memory fault, a floating point exception, or incorrect results.

MAD is more than just a debugging environment, it is also a generic playback tool that can be used to improve and modify the implementation of Mentat objects as well as perform optimizations. The power of MAD lies in its playback capabilities and its ability to let programmers focus on a single object and execute it in complete isolation from the rest of the system.

MAD is simple to use and requires little intervention. To start the recording phase literally requires the addition of one word on the command line when invoking an application. For the playback phase, programmers use a point-and-click interface to select a specific object for debugging (Figure 3). For more information about the implementation of MAD please see [18].

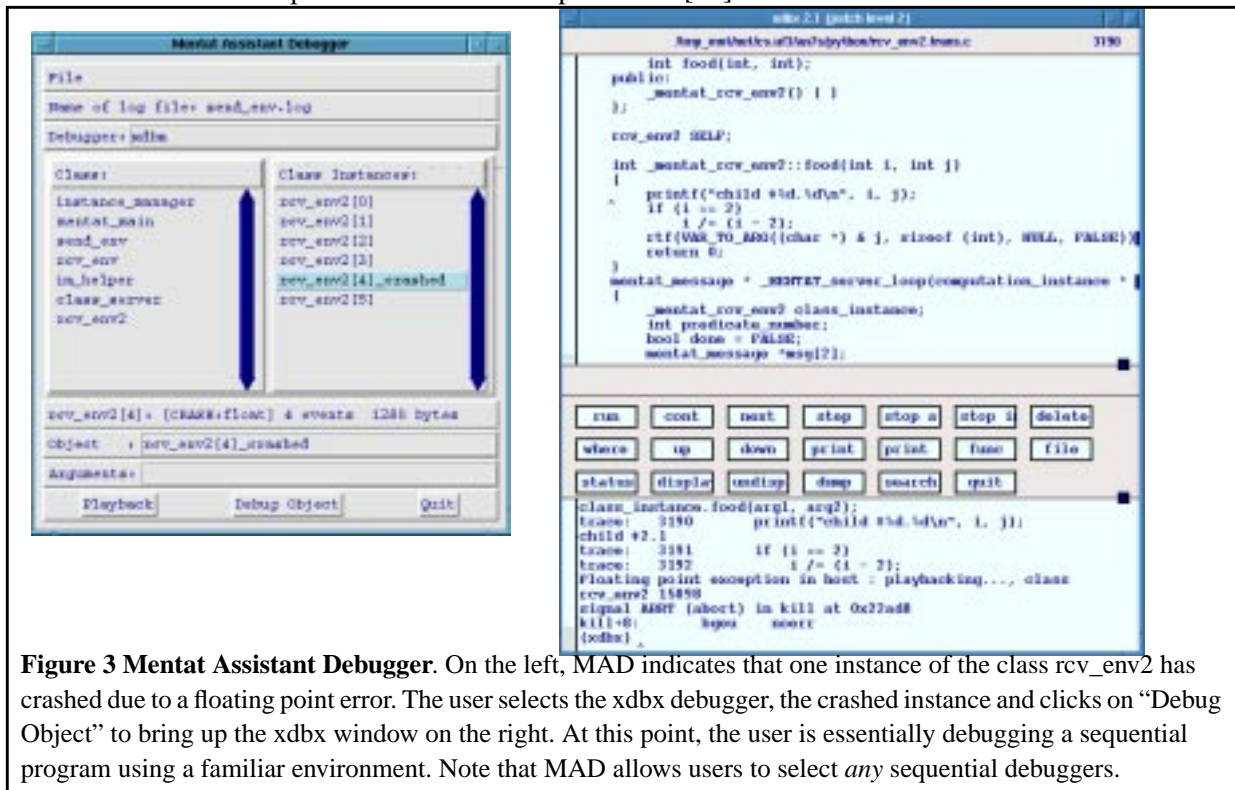


Figure 3 Mentat Assistant Debugger. On the left, MAD indicates that one instance of the class `rcv_env2` has crashed due to a floating point error. The user selects the `xdbx` debugger, the crashed instance and clicks on “Debug Object” to bring up the `xdbx` window on the right. At this point, the user is essentially debugging a sequential program using a familiar environment. Note that MAD allows users to select *any* sequential debuggers.

3.6. Support for other programming models: PVM

Clearly we cannot expect all Legion users to use the Mentat Programming Language (MPL); other parallel processing languages and models must be supported. This is particularly true for legacy codes. Our plan for legacy code and multiple model interoperability is detailed in [8]. One of the early tests for our plan is support for PVM [21], a message passing parallel processing system that is a *de facto* standard.

At first glance, it seems counter-intuitive to implement PVM on top of the CWVC which provides a higher level of abstraction. We have implemented PVM on the CWVC so that existing PVM applications can be executed in the CWVC environment without being rewritten. The ported PVM codes not only execute, they benefit from CWVC features such as the federated file system, pain management, and our load-sensitive scheduling algorithms. The ability to use existing codes, and later to integrate them with other parallel codes, supports our larger Legion goals of legacy code support and multi-language interoperability.

The PVM implementation is straightforward. Each PVM instance is represented by a typed Mentat object. Calls such as `pvm_initiate()`, `pvm_send()`, and `pvm_rcv()` are implemented using calls to the CWVC

run-time system that instantiates objects, sends messages, receives messages, etc. PVM-specific services, (e.g., barriers), are implemented using synchronization objects.

To test the correctness of our implementation and to compare its performance with the standard “native” PVM implementation, we took two existing PVM applications and ran them in the same environment. The two applications were the latency/bandwidth test code that comes with the PVM distribution, and the PVM implementation of the NAS benchmarks [5]. Table 2 presents the performance comparison between the

Message size (bytes)	Native PVM time (mSec)	Mentat-PVM time (mSec)	Native PVM bandwidth (bytes/Sec)	Mentat-PVM bandwidth (bytes/sec)
100	8.81	8.47	11,800	13,310
1000	9.88	9.75	101,269	102,550
10,000	27.52	26.58	363,380	376,220
100,000	194,231	196,783	514,860	509,880
1,000,000	1,920,624	1,951,618	520,670	512,790

Table 2 Point-to-point benchmarks on 40 Mhz Sparc 2’s. All measurements are on an 8 processor cluster using raw (no XDR) encoding, and direct routing. The time is the time to send a message of the specified size and to receive a one byte reply. It is the average of 3 runs (each run being the average of 20 sends/receives)

native PVM and Mentat-PVM implementations on the communication benchmarks. For short messages ($\leq 10,000$ bytes) the Mentat-PVM implementation outperforms the native implementation (this is particularly encouraging since most messages in many applications are short). The better Mentat-PVM time is the result of the optimized communication system used by Mentat.

To test application performance, we selected the PVM NAS benchmark implementation because it is familiar and accepted. Table 3 presents the performance results.

Application	Time (Sec)		Communication Time (Sec)		Communication Volume (MB)	
	Native PVM	Mentat-PVM	Native PVM	Mentat-PVM	Native PVM	Mentat-PVM
IS kernel	226	256	202	207	140	140
EP kernel	346	350	NA	NA	NA	NA
MG kernel	123	110	62	59	49	49

Table 3 NAS Benchmark Results. The measurements were taken using eight 40 Mhz Sparc 2’s, using raw encoding. The IS kernel sorted 221 keys in the range [0..219], the EP kernel problem size was 226, and the MG kernel problem size was 128.

4. Results

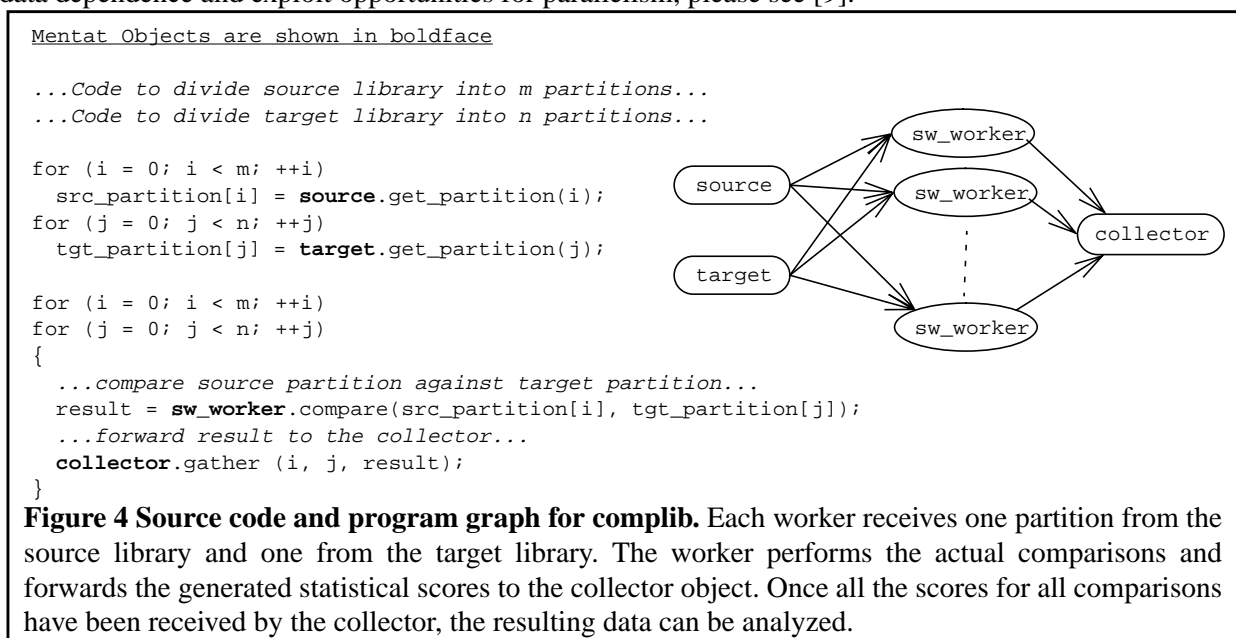
The bottom line for users is application performance. Below we present results for one of the many applications operating on the CWVC. We describe the application, the test environment, and the results.

Biochemists are often interested in the three dimensional structure of a new protein to determine its function. A commonly used technique compares the new protein’s sequence of amino acids with those of known proteins and obtains a statistical score that reflects their commonality. Three popular algorithms for performing the sequence comparisons are Smith-Waterman [20], FASTA [4], and Blast [1]. The latter two are heuristics; the quality of the score is traded for speed. Smith-Waterman is the benchmark algorithm, generating the most reliable scores, but at a considerable time expense.

Our test application, *complib*, compares two protein or DNA sequence libraries using the Smith-Waterman algorithm. Each sequence is represented by a name and a variable string of characters, also known as residues. Every sequence in the source library is compared against every sequence in the target library. For each sequence in the source library, statistics are generated on how the sequence compares to

the target library as a whole. Note that each sequence comparison can be performed independently, making this application particularly suitable for a parallel implementation. The program is written in the Mentat Programming Language and consists of several objects: the source library, the target library, a collector object to monitor the progress of the computation, and workers to perform the actual comparisons. The heart of the application and its corresponding program graph are shown in Figure 4. The algorithm first divides the source library into m partitions and the target library into n partitions. In the main “for” loops, the algorithm instantiates $m \times n$ workers to perform the sequence comparisons. Each worker then reports its results to the collector object. Once all the statistical scores have been generated, the program sorts and analyzes the data.

The Mentat compiler transforms the user’s source code and interacts with the underlying run-time system to dynamically detect data dependence and transparently build program graphs. Mentat automatically exploits the fact that the workers are stateless objects⁵ and thus the system may create new instances as needed. Furthermore, Mentat detects that there are no data dependencies between workers and issues concurrent scheduling requests for all workers. Another advantage of using stateless objects is that the system scheduler dynamically load-balances regular object requests across the system. This feature is especially important in a shared heterogeneous environment where processing capabilities can vary greatly and where the CPU resource usage pattern is highly variable. For more information on how Mentat detects data dependence and exploit opportunities for parallelism, please see [9].



The computing resources available to the CWVC consist of approximately one hundred hosts from major workstation vendors (IBM, SUN, SGI, HP) and include an 8-node IBM SP-2. These hosts are located in six different buildings, each of which has one or more Ethernet segments that are connected to the university fiber-optic backbone. The hosts do not share a single file system, but users can transparently access their files and binaries by using the Federated File System.

We measured the elapsed wall clock time for complib using a 20 sequence source library containing 4478 residues and a 10,716 sequence target library consisting of 3,647,403 residues. We report performance in millions of matrix entries per second (MEPS) which is a standard unit of measure for benchmarks in the biochemistry community. The number of matrix entries is obtained by multiplying the number of residues in the source library with the number in the target library. Our experiment contained approximately 16,333 million matrix entries.

⁵. Users may declare Mentat class as *regular*, meaning that it is logically stateless.

To provide a benchmark for comparison we executed a sequential version of complib on all platforms for which complib is available. Table 4 lists the sequential execution times (CPU user time), the corresponding MEPS, and the set of participating hosts for complib.

Platforms	Sequential Performance		Available CWVC Hosts for Complib	
	Time (sec)	MEPS	Quantity	Max. Number of Processors
Sparc Ultra	4003	4.08	3	1
Sparc 20	9756	1.67	4	4
Sparc 10	13460	1.21	5	4
Sparc LX	23146	0.71	2	1
Sparc 2	33823	0.48	8	1
RS6000 / 250 ^a	31833	0.51	9	1
SGI Reality Engine	7487	2.18	1	2
SGI Indy	11386	1.43	12	1

Table 4 Sequential performance and available CWVC hosts for complib

a. A mix of RS6000's were used, including seven SP-2 nodes.

Table 5 reports the parallel execution times on the CWVC. The wall clock times represent 22 runs and do not include the start-up overhead of initializing the source and target libraries. We divided both libraries into partitions of 10 sequences so that each worker was responsible for 100 sequence comparisons. In total, there were 2144 requests made to the workers per run.

Elapsed Time (sec)			MEPS		
min	max	mean \pm std	min	max	mean \pm std
251	321	274 \pm 16	51	65	60 \pm 3.3

Table 5 Performance of complib on the CWVC

5. Problems encountered

We encountered several problems transforming Mentat into the CWVC. Some of these, such as the need for a federated file system and single name space were anticipated. Others were not. At least three of these problems are not Legion specific, they will need to be overcome by most systems that have the same objectives as Legion/CWVC.

The first problem is the trend in workstation operating systems towards the exclusive use of dynamic linkers. For example, Irix, Solaris, and AIX do not support static linking. This is a problem whenever hosts do not share the same file system structure, in other words object libraries may be at a different location on different hosts, or not present at all. Thus, an executable that executes on one host, will not execute on another host of the same architecture type. This limits our ability to transport binaries from one location to another. Executable transport is not an issue in older operating systems such as SunOS which permit static linking. Another facet of this problem occurred when we implemented the stdio library call traps. We could not statically link our routines in to replace the underlying C library routines. To solve this problem required that we explicitly manage the dynamic linker in our code. Unfortunately the mechanisms required vary from system to system.

A second class of problems that we encountered relate to Unix itself. It is not sufficient to simply scale the number of hosts when using a Unix based parallel processing tool if there is a single host that starts remote shells executing on other hosts. For example, if a daemon on host A starts daemons on hosts B..Z. This technique, which we used to practice, and which PVM uses begins to fail when there are many hosts

for at least two reasons. First, the number of open files can rapidly exceed the limits of the operating system. (These limits can be changed by recompiling the operating system.) The problem is that each rsh consumes at least two file descriptors, and possibly more if the pipe() command is used as well. Even if these file descriptors are closed by the “main” daemon they are not always closed immediately as called for in the manuals. Instead, there is usually a time-out of just under five minutes. Under normal operating conditions this is not a problem, but when a very large number of rsh’s are being generated in a short period of time, such as when starting the parallel system up, the system may run out of descriptors.

A related problem has to do with the use of NFS file servers. Simultaneous execution of a large number of copies (> 60) of the same executable, as in a data parallel program or system start-up, may overload the file server, resulting in multiple lost requests. The host operating systems treat this as an error, and report that the executable does not exist, when clearly it does.

A final problem we encountered is application fault-tolerance. While the system itself is fault-tolerant and recovers from host failure, applications do not. If my application has an object on a host that has failed, then my application blocks, and never recovers. This requires the user to kill and restart the application. We consider this unacceptable in the long run. We have implemented fault-tolerance in two applications at the application level, and are exploring general application fault-tolerance.

6. Related work

Workstation cluster management systems are typically either throughput oriented or response-time oriented. Throughput oriented systems attempt to exploit available resources to service the largest number of *jobs*, where a job is a single program that does not communicate with other jobs. DQS [6], Condor [15], LoadLeveler [12], and NQS [14] are throughput oriented systems [13]. Response time oriented systems attempt to minimize the execution time of a single application by harnessing the available workstations to act as a virtual parallel machine. Examples of such parallel processing tools available for workstations include Linda [3], Mentat [7][22], PVM [21] and MPI [17]. Legion is both a response time and a throughput oriented system. Unlike the above systems Legion is a complete, comprehensive system, and tackles problems such as fault-tolerance, security, and scalability.

The vision of a seamless metacomputer such as Legion is not novel; worldwide computers have been the vision of science fiction authors and distributed systems researchers for decades. However, to our knowledge no other project has the same broad scope and ambitious goals of Legion. Fortunately, it is not necessary to develop all of the required technology from scratch. A large body of relevant research in distributed systems, parallel computing, fault-tolerance, management of workstation farms, and pioneering wide area parallel processing projects, provide a strong foundation on which to build.

Related efforts such as OSF/DCE [16] and CORBA [2] are rapidly becoming industry standards. Legion and DCE share many of the same objectives, and draw upon the same heterogeneous distributed computing literature for inspiration. Consequently, both projects use many of the same techniques, e.g., an object-based architecture and model, IDL’s to describe object behavior, and wrappers to support legacy code. However, Legion and DCE differ in several fundamental ways. First, DCE does not target high-performance computing; its underlying computation model is based on blocking RPC between objects. Further, DCE does not support parallel computing; instead, the emphasis is on client-server based distributed computing. Legion, on the other hand, is based upon a parallel computing model, and one of our primary objectives is high performance via parallel computation. Another important difference is that Legion specifies very little about the implementation. Users and resources owners are permitted—even encouraged—to provide their own implementations of “system” services. Our core model is completely extensible and provides choice at every opportunity—from security to scheduling to fault-tolerance. Similarly, CORBA [2] defines an object-oriented model for accessing distributed objects. CORBA includes an Interface Description Language, and a specification for the functionality of runtime systems that enable access to objects (ORB’s). But like DCE, CORBA is based on a client-server model rather than a parallel computing model, and less emphasis is placed on issues such as object persistence, placement, and migration.

7. Summary

Legion is an ambitious middleware project that will provide a solid, integrated, conceptual foundation on which to build applications. One could argue that Legion is perhaps too ambitious, that there are just too many different complex issues to address. Tackling too many different issues is certainly a risk. On the other hand, eventually there will be Legion-like metasystem software; it is a necessary condition for a large scale digital society. The real issue is whether it will come about by design, in an organized and coherent fashion, or by pasting together different solutions.

Legion, as defined by our objectives, is not yet a reality. Rather than attempting to construct Legion from scratch we have chosen to begin with an existing system, Mentat, and transform it into Legion by incorporating research results from over twenty years of heterogeneous distributed computing. The first step in the transformation is the construction of the campus-wide virtual computer at the University of Virginia. The objectives of the campus-wide virtual computer 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,
- act as a testbed for the worldwide Legion,
- permit application development in parallel with Legion system development.

The prototype implementation is well on its way to meeting these objectives. The performance results provide evidence that workstation based, heterogeneous parallel processing can be used to solve computationally challenging problems of interest to university researchers at reduced cost. With respect to the testbed goal, the CWVC has been an invaluable tool which has enabled us to begin trying out designs and stressing implementations with real applications. Our experience putting the CWVC together highlighted several critical factors that must be addressed in both the short term and in the long term.

In March of 1996 we began our implementation of the core Legion object model. Unlike the existing CWVC prototype the new implementation incorporates mechanism for security, fault-tolerance, application directed scheduling, autonomy, scalable binding, etc. This “full blown” implementation re-uses many components of the prototype (e.g., the compiler and debugger) but for the most part is being written from the ground up. We expect to have a usable documented system available for public use in mid 1997. The system, and sources, will be publicly available.

The new implementation will be used in several projects. Legion is the metasystem software being used in the DARPA funded Distributed Object Computation Testbed project, a collaborative effort between SDSC, NCSA, SAIC, and the University of Virginia. We are also collaborating with researchers at Sandia National Labs and Los Alamos National Labs on Legion support for the ASCI (Accelerated Strategic Computing Initiative), as well as with several sites in the NSF Partnerships for Advanced Computational Infrastructure competition.

8. Acknowledgments

We would like to thank Bill Pearson of the biochemistry department for introducing us to sequence comparison, and for collaborating on the parallel version. We would also like to thank all of the members of the Legion team. The faculty are Bill Wulf, Jim French, Paul Reynolds Jr., and Alf Weaver. Staff members are Mark Hyett and Lindsey Faunt. The students who have worked on various components are Jon Weissman and Anh Nguyen-Tuong (run-time system), John Karpovich, Matt Judd, and Adam Ferrari (federated file system and I/O libraries), Emily West (complib) and with Mark Morgan (pain management and thermostat), Chenxi Wang (security), Roger Harper (PVM), and Mike Lewis. We would also like to thank the Jet Propulsion Lab for the use of their Intel Paragon.

9. References

- [1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, "Basic local alignment search tool", *J. Mol. Biol.*, 215, pp. 403-410, 1990.
- [2] Ron Ben-Naten, "CORBA: A Guide to the Common Object Request Broker Architecture," McGraw-Hill, 1995.
- [3] N. Carriero and D. Gelernter, "Linda in Context," *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April, 1989.
- [4] A. S. Deshpande, D. S. Richards, and W. R. Pearson, "A platform for biological sequence comparison of parallel computers", *CABIOS*, 7, pp. 237-247, 1991.
- [5] A. J. Ferrari, A. Filipi-Martin, and S. Viswanathan, "The NAS Parallel Benchmark Kernels in MPL", University of Virginia Computer Science Technical Report CS-95-39, September 12, 1995.
- [6] T. P. Green and J. Snyder, "DQS, A Distributed Queueing System," Florida State University, March 1993.
- [7] A. S. Grimshaw, "Object-Oriented Parallel Processing with Mentat," *Informatics and Computer Science*, 1996.
- [8] A. S. Grimshaw, W. A. Wulf, J. C. French, A.C. Weaver, and Paul Reynolds Jr. "Legion: The Next Logical Step Toward a Nationwide Virtual Computer," Computer Science Technical Report, University of Virginia, CS 94-21, June, 1994.
- [9] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer, "Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing," *ACM Transactions on Computer Systems*, pp. 139-170, Vol. 14, No. 2, May 1996.
- [10] A. S. Grimshaw and W. A. Wulf, "Legion—A View from 50,000 Feet," *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, Syracuse, NY, August 6-9, 1996.
- [11] P. J. Hatcher, et al, "Data-Parallel Programming on MIMD Computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 3, pp. 377-383.
- [12] International Business Machines Corporation, "IBM LoadLeveler: User's Guide.", Kingston, NY, March 1993.
- [13] J.A. Kaplan and M.L. Nelson, "A Comparison of Queueing, Cluster, and Distributed Computing Systems," NASA Technical Memorandum 109025, NASA LaRC, October, 1993.
- [14] B. A. Kingsbury, "The Network Queueing System," Palo Alto, CA, March 1993.
- [15] M. Litzkow and M. Livny, "Experience with the Condor Distributed Batch System", *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL, October, 1990.
- [16] H.W. Lockhart, Jr., "OSF DCE Guide to Developing Distributed Applications," McGraw-Hill, Inc. New York 1994.
- [17] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," May 1994.
- [18] A. Nguyen-Tuong, A. S. Grimshaw, "Exploiting Sequential Debuggers in a Parallel Processing Environment: An Introduction to the Mentat Assistant Debugger," *Computer Science Technical Report CS-95-22*, University of Virginia, 1995.
- [19] A. Nguyen-Tuong, A.S. Grimshaw, M. Hyett, "Exploiting Data-Flow for Fault-Tolerance in a Wide-Area System," *To be published in the proceedings of the 15th International Symposium on Reliable and Distributed Systems*, 1996.
- [20] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences", *J. Mol. Biol.*, 147, pp. 195-197, 1981.
- [21] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.
- [22] G. Wilson and P. Lu, "Parallel Programming Using C++," The MIT Press, 1996.