Interoperability of Parallel Systems: Running PVM Applications in the Legion Environment[1]

Roger R. Harper
May 5, 1995

1. This work was supported by NSF grants ASC-9201822 and CDA-8922545-01.

1.0 Introduction

Parallel programming systems based on loosely coupled networks of computers are becoming more and more popular. Although they may not have as fast a communication system as that present in massively parallel processors (MPP's), networks of computers are desirable because of their availability, their cost-to-performance ratio and the rich set of development and support tools that are available.

Although there is a large set of available tools for developing parallel programs for networked computers, these tools are by and large incompatible with each other. This means that a user must be certain that the development environment that was used in developing his/her application is present on every system that he/she intends to use. This inevitably restricts developers in the resources (available computers) that they may use, in that, even though a more powerful computer is available, the programmer may not be able to use it because the development system that he/she uses is not available on that machine.

This project - the development of a software library that will allow programs written for version 3 of the Parallel Virtual Machine software system to execute in the Legion environment - is an attempt to provide a solution to that problem. This project is important for a two main reasons. First, as explained above, a library of this kind provides developers with an opportunity to exploit resources that previously may have been inaccessible to them. Secondly, this library meets one of the goals of the Legion system to be able to support other parallel programming models, in this case PVM, and to be able to execute application programs written for those models.

In addition, by running in their applications in the Legion environment, developers are able to take advantage of Legion features that may not be present in their native systems. These features include accounting and resource management mechanisms, improved scheduling abilities, improved protection and security, a distributed file system that presents a singe, persistent namespace to applications, and improved fault-tolerance. (For a complete description of these Legion features see [9] & [10].)

This document is divided into the following sections: Section 2 provides brief descriptions of both the PVM and Legion systems, Section 3 describes the design of the software library, Section 4 examines the performance of the library software, Section 5 looks at PVM features that were not included in the library, and Section 6 describes how to use the library software.

2.0 The Parallel Virtual Machine and Legion Systems

This section of the document gives a small description of the features and system architectures of the PVM and Legion parallel programming systems.

2.1 Parallel Virtual Machine

The Parallel Virtual Machine (PVM) is a "... programming environment for the development and executing of large concurrent or parallel applications" [1]. It was developed as a result of a collaborative effort between researchers at Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University. PVM allows an application developer to think of a set of heterogenous (or homogenous) computers as a single parallel virtual machine. PVM transparently handles all message routing, data conversion, and task scheduling across the network of computer architectures.

There are two parts to the PVM system, a dæmon process that resides on each of the computers that make up the virtual machine, and a library of interface routines that the user links in with his/her application program. The user is able to access the PVM resources by means of these routines. In addition, these routines allow for the creation and termination of tasks, as well as communication and synchronization between tasks. The communication routines include the "usual" ones that send and receive messages to and from a certain task, as well as routines that allow for multicasting and broadcasting to a group of tasks. The synchronization routines provide the ability to perform barrier synchronizations as well as global operations. Figure 1 presents a simplified view (adapted from [5]) of the PVM system architecture.

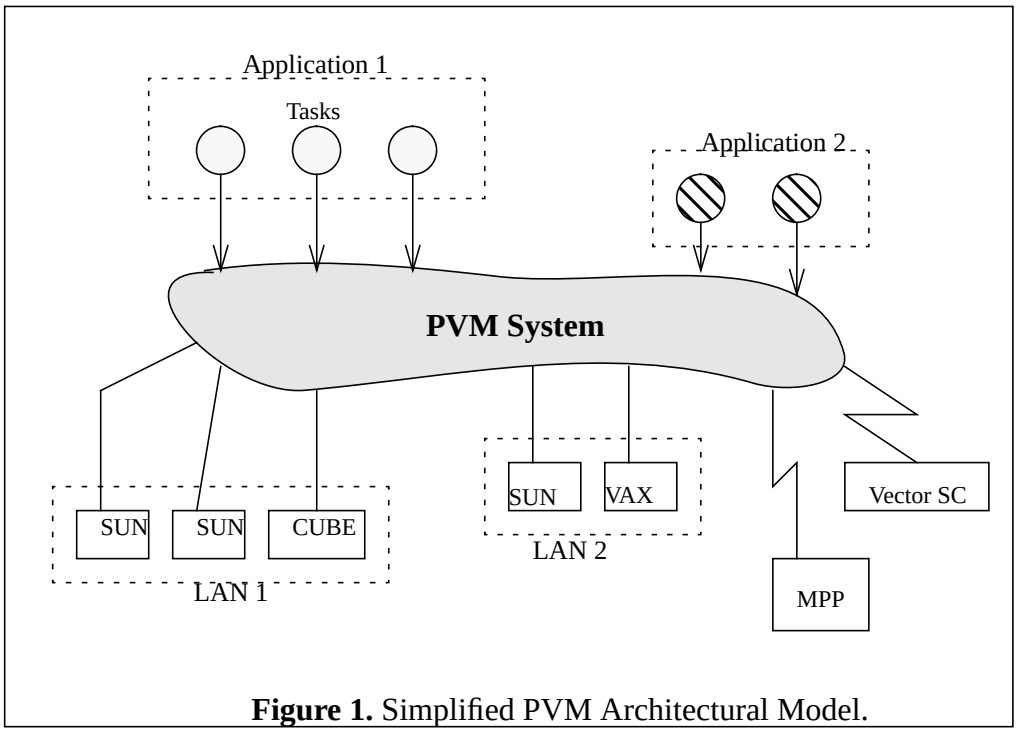

Figure 1. Simplified PVM Architectural Model.

A user develops his application as a collection of cooperating *tasks*, identified by a unique task ID (TID), and each task is responsible for a part of the application's computations. The PVM system currently supports applications written in C, C++, or Fortran. As a result of its simplicity of, and the rich set of features that it provides, "... PVM system has gained widespread acceptance in the high-performance scientific computing community" [2].

2.2 The Legion Project

The principal goal of the Legion project is to combine the communication and computation resources already available into a nation-wide virtual computer. Legion is intended to be made up of workstations, and vector and parallel supercomputers connected by local networks and by the national information infrastructure. The benefits of having a system such as Legion include: (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 [10].

Legion will be built on the foundation that was laid by Mentat [3]. This approach has two major advantages. First, it will reduce the amount of new code that must be written before Legion can be used, and second, it provides Legion with a foundation that is known to work well. In addition, the Legion architecture is an object-based one in which objects have a class and communicate via messages. The architecture is implemented as a layered virtual machine with a scalable, distributed control [10]. Figure 2 presents a simplified view of the Legion architecture.

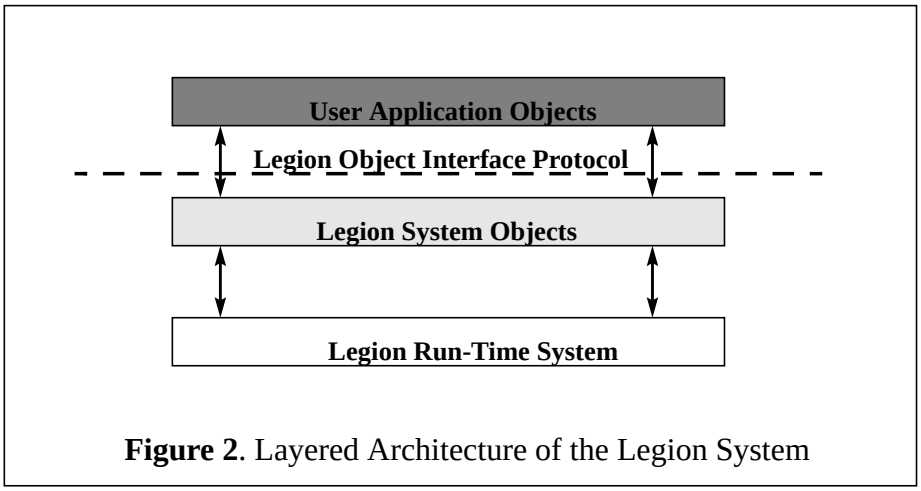

Figure 2. Layered Architecture of the Legion System

Although Legion is designed using the object-oriented paradigm, it is imperative that it provide support for applications that are written in languages, such as Fortran, that are not object-oriented. Legion intends to provide this support in one of three ways [10]: (1) by implementing “wrappers” for applications written in other languages; (2) by exporting the Legion run-time system interface and retargeting existing compilers, and (3) by a combination of the two.

In the first alternative, the original application code is encapsulated in a function that is written in a language that Legion directly supports (for example the Mentat programming language, MPL). In this way, the original application becomes nothing more

than another member function in a Mentat class and so it can interact with Legion is via the regular Legion object interface protocol (Figure 3a).

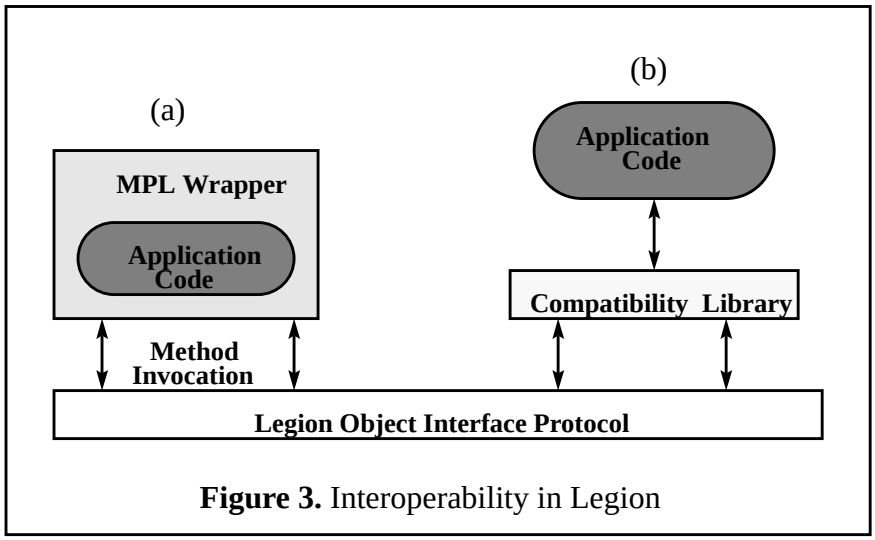

Figure 3. Interoperability in Legion

If the second option is used, compilers for other parallel processing systems could be redesigned in one of two ways. First, they could be changed so that application are provided with direct “hooks” into the Legion run-time system, or second, they could provide compatibility libraries that emulate the native services with Legion services (Figure 3b).

2.2.1 Mentat Features Used in Legion

Mentat was “... designed to directly address the difficulty of developing architecture-independent parallel programs” [3]. Its objectives are to provide easy-to-use parallelism, to achieve high performance through parallel execution, and to permit the execution of applications across a wide range of platforms. Mentat aims to provide high-level abstractions that “... mask the complex aspects of parallel programming, communication, synchronization, and scheduling, from the programmer”[3]. There are two main parts of the Mentat system, the Mentat Programming Language (MPL) and the Mentat run-time system.

MPL is based on the C++ language, and in much the same way as a regular C++ class, each Mentat class consists of local and member variables and member functions. In

addition, however, each Mentat class also has its own thread of control. The Legion system objects are coded using MPL The Legion run-time system is currently composed of three dæmons: phoenix (the current fault-tolerance dæmon), the instantiation manager (IM), and the token matching unit (TMU). The IM and TMU are Mentat dæmons that have been adapted for use in Legion.

MPL is one of the languages that is currently supported by Legion. It is based on the C++ language, and in much the same way as a regular C++ class, each MPL class consists of local variables and member functions. MPL classes are those whose member functions the programmer thinks are computationally intensive enough to be worth doing in parallel. As such, each MPL class has its own thread of control. The Mentat class is designated by the use of the **`mentat`** keyword. The Mentat compiler detects and manages invocation of, communication between, and synchronization among Mentat objects (instances of Mentat classes are called Mentat objects).

3.0 Design of the Library Software System

The library system is composed of a set of routines that interact with the Legion run-time system. For each PVM routine, there is a corresponding library routine that provides an identical interface and response. When an application makes a call to a PVM function, the call goes through the library and is serviced by the Legion run-time system instead of the PVM system (as described in section 2.2). Figure 4 illustrates this process.

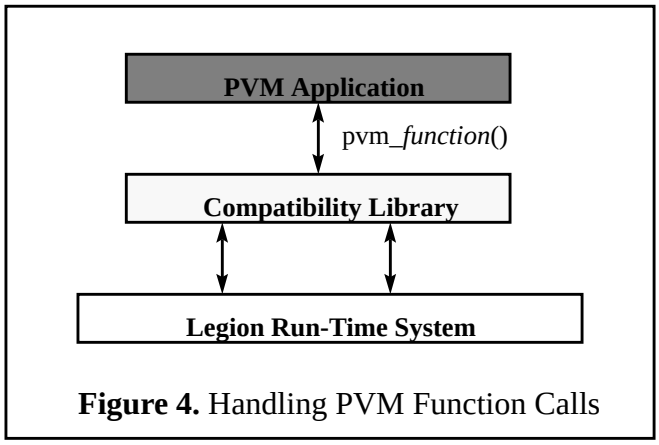

Figure 4. Handling PVM Function Calls

The interfaces to the PVM functions are unchanged, so that all that is required to run current applications is a recompilation (see Section 6). This section describes the design of the library software system, and the factors that influenced the design decisions.

3.1 Process Identification

As was stated earlier, each process in PVM has a unique TID. This ID is a 32 bit integer that contains information that the PVM system can extract and use. The Mentat system also provides a name to objects. Each Mentat object has a unique name that is currently composed of two integer parts, a host number (which host is the object on) and a port number (which port is the object using for communication). The library software uses the Mentat object name to assign unique names to PVM applications. The TID that is assigned by the library software is a 32-bit integer that is made up of the object name's host number in the upper 16 bits, and the port number in the lower 16 bits (see Figure 5).

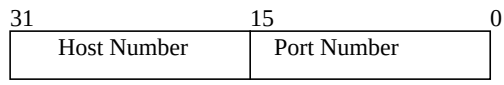

Figure 5. Format of the Library Assigned TID

The mapping between object names and TID's, and vice versa, are accomplished by means of two functions:

int encodetid(object_name); and
object_name decodetid(int);

The first function takes a Mentat object name and returns the corresponding TID, while the second function takes a TID as its parameter and returns the matching Mentat object name.

As a consequence of building TID's from Mentat object names, the library is restricted to a Legion environment of at most 65,535 hosts and to port numbers smaller than 65,535. This should not be a problem, however, since it is unlikely that in practice a Legion system will have that many hosts. It should also be noted that the Legion identifi-

cation method is currently being redesigned to a format that is different from the current (host-number, port-number) format to a new Legion ID. The new Legion ID's will be a 128-bit entity that will have system information embedded within it. This change will have to be reflected in a redesign of the **encodetid()** and **decodetid()** functions to allow them to build TID's based on the new Legion ID's.

Instead of building the TID's from Mentat object names, alternative approaches that were considered included: (1) to have each task keep a complete list of TID's and the corresponding object names, and (2) to have a centralized repository of TID's and object names. Neither of these alternatives would have been a good choice. The first alternative would necessitate the sending extra messages to every task each time a new task is created or destroyed so that tasks could invalidate the TID's in their list. This would have had negative, and unacceptable, repercussions on the performance of the system. The second alternative would have introduces a potential bottleneck and a "single point of failure" into the system, and is not scalable to large numbers of processes or hosts. This too, would have been unacceptable.

By choosing to build the TID's from the object names, the library functions are able to translate from one into the other, thus removing the need for any type of lists. Consequently, the system is quite scalable and more distributed (task management functions are kept as local to the tasks as possible), and has good performance (no need to perform any table lookups).

3.2 Process Control and Information

One of the more important features of PVM, and thus of the library software, is the ability to create and manage new processes (tasks). New tasks are created by means of the **pvm_spawn()** function. The library implements this function as a call to the **instantiate()** function of the IM. Consequently, when no location hints are used in the call to **pvm_spawn()**, the library is able to use the scheduling algorithm of the run-time system to create and place the new task on the host that is best suited for it. When a specific host is given as a location hint to **pvm_spawn()**, the new task is created there, whereas, if an archi-

tecture type is used, the new task is placed on the host (of the architecture type) with the lowest CPU load.

When a task makes its first call to a library function, it is registered with the IM and assigned a TID. In addition, resources that the library requires (for example, memory for lists and message buffers) are allocated and the task gets its parent's TID. The **pvm_exit()** function releases any resources held by the process, and breaks the "connection" between the process and the Legion run-time system (by unregistering with the IM, calling **mentat_cleanup()** (a Legion system function), and clearing the socket connection tables.).

The **pvm_kill()** function, used to kill another task, also makes use of the functionality of the IM. This function (**pvm_kill()**) first maps the given TID into a corresponding Mentat object name, extracts the host from the object name, then uses the **set_node_id()** and **kill_object()** functions of the IM to kill the process.

The PVM process information routines are also implemented by making calls to relevant functions of the IM. **pvm_config()**, which returns information about the configuration of the virtual machine, and **pvm_mstat()**, which determines the status of a specific host, both make a call to the IM's **get_name_map()** function which returns information about the current Legion system. The **get_name_map()** function returns a **name_map_rec_list** structure. This structure is a list of **name_map_rec** structures (Figure 6a), one for each host in the system. The information from these structures is then mapped into the appropriate fields of the **pvmhostinfo** structures (Figure 6b) that are passed as parameters to **pvm_config()**. The **`host_id`** field is mapped into the string name of the host, the **`hosttype`** field is mapped into the string name of the host's architecture type, and the **`mipsrating`** field is used as the relative speed of the host's CPU. **pvm_mstat()** simply looks at the **`active`** field of the name_map_rec structures. This field indicates whether the host is up or not.

The **pvm_pstat()** function, which determines the status of a specific task, first converts the TID, that was passed as a parameter, into the corresponding Mentat object name. It then gets the host that the process is on (from the object name) and makes a call to the

```
enum supported_host_type {is_sun4, is_hp, is_solaris, is_sgi,
                          is_rs6000, is_gamma);
struct name_map_rec {
  int                  host_id;
  int                  active; // Is this host active or not?
  int                  mipsrating;
  int                  mflopsrating;
  supported_host_type  hosttype;
  int                  processors;
  char                 host_name[32];
};
```

(a)

```
struct pvmhostinfo {
    int         hi_tid;    // TID of PVM daemon on this host.
    char        *hi_name; // Host name.
    char        *hi_arch; // Architecture type.
    int         hi_speed; // Host's relative CPU speed.
};
```

(b)

Figure 6. Host Information Structures of Legion and PVM

IM's **get_instances()** function, which returns a list of **instance_rec** structures (Figure 7a), each one representing an active Legion processes on the host. This list is then searched for a matching object name. (It should be noted that the current implementation of **get_instances()** does not indicate whether the processes are "hung". Improvements to this function are being made.)

The **pvm_tasks()** function also makes use of the **get_instances()** function. (**pvm_tasks()** returns information about all the tasks that are running in the virtual machine.) In this case, however, the information that is returned from the call to **get_instances()** is used to fill in the fields of the **pvmtaskinfo** structures (Figure 7b) that **pvm_tasks()** returns. The **d_name** field of the **instance_rec** structure is mapped into the **ti_tid** of the **pvmtaskinfo** structure. The **d_name** field also provides the number of the host that the task is on. This host number is then combined with the port number that the IM uses, to get the **ti_host** field. The **obj_class_name** and **u_pid** fields provide the executable name and process PID that go into the **ti_a_out** and **ti_pid** fields respectively. Unfortunately, there is no information

```
enum state_type {suspend, suspend_all, resume, user_prog};
struct instance_rec {
  object_name         d_name; // Object name of this process.
  int                 in_use;
  int                 u_pid;
  char                obj_class_name[20]; // Executable name.
  state_type          state;
  long                last_time_used;
  int                 access_cnt;
  float               percent_requested, percent_set,
                      percent_used, mem_set;
  int                 num_calls, data_volume;
  timeval             cpu_use;
  #ifdef M_VIRT_HOST
  int                 node;
  #endif
};
```

(a)

```
struct pvmtaskinfo {
   int        ti_tid; // Task TID.
   int        ti_ptid; // Parent's TID.
   int        ti_host; // TID of the daemon on this task's host.
   int        ti_flag; // Task's status flags.
   char       *ti_a_out; // Executable's name.
   int        ti_pid; // Process PID.
};
```

(b)

Figure 7. Legion and PVM Task Information Structures

that corresponds to the **`ti_ptid`** and **`ti_flag`** fields, and so these are always returned with a value of 0 (see Section 5).

3.3 Communication

Communication in the library system, as well as in PVM, is accomplished by means of message passing. The software was designed so that message passing would be as efficient as possible, both in terms of time and space. Again, as in other facets of the library, the real work of sending and receiving messages is left to the Legion run-time system. In addition, the library borrows heavily from the original PVM system when it comes to the packing and unpacking of messages.

The library's messages are stored in buffers that are allowed to grow dynamically as more data is packed into it. The class that is used to hold a message is defined in Figure

8a. This has fields for each part of the message header, as well as a pointer to the actual

```
class pvm_message {
 public:
    int           srctid;         // Source TID.
    int           wait_id;        // Wait ID for this message.
    int           tag;            // Message tag.
    int           length;         // Message length (in bytes).
    int           encoding;       // Message Encoding format.
    int           data_sz;        // How much space for message.
    int           unpkpos;        // Where to start unpacking data.
    char          *msg_data;      // Message data buffer.
    XDR           xdr_ptr;        // XDR pointer for this message.
    pvm_message   *next;
    pvm_message   &operator=(pvm_message &);
};
```

(a)

(b)

Figure 8. Message Structure and Buffer Definitions

data that will make up the message. The message header provides information that the receiving task uses to identify the origin of the message. The message buffer is originally allocated to be ***PvmFragSize*** bytes (by default this is 4096 bytes), but as was stated above, it is allowed to grow as needed.

When the message is being sent, the message structure is compressed into the format shown in Figure 8b. The message header is thirty-two (32) bytes long and contains the source TID, the wait ID, the message tag, length, encoding, and an indication of how much space must be allocated for the data by the receiver. (Since the data buffer is allowed to grow dynamically, the receiver must know how much space is needed to store it, and the length of the message would not always give a correct indication. For example when strings are being sent, the buffer also contains an integer that indicates the length of the string.)

After the message has been compressed, it is stored in a **mentat_message** structure, and passed on to the Mentat run-time system to be sent to the specified task. (Since the run-time system sends the messages, they must be packed into a **mentat_message**.). The

messages are then sent by means of the run-time system's **invoke_fn()** function. This function is asynchronous, so that processing is able to continue as soon as possible.

Messages are received (**pvm_recv()**) by making a call to the run-time system's **block_predicate()** function. This is a blocking function call, that waits until a message has been received before processing is allowed to continue. When a message is received, the steps that were taken to send the message are done in reverse. A **mentat_message** structure is received, from which a buffer (of the form in Figure 8b) is extracted. A **pvm_message** structure (as in Figure 8a) is then built from the fields in the buffer. If this message is not the one that was expected, it is placed in a message queue and the receiving task blocks and waits for another message to arrive. (The message queue is actually a linked-list of **pvm_message** objects.)

Although the **block_predicate()** function is a blocking one, non-blocking receives (**pvm_nrecv()** and **pvm_trecv()**) are also implemented in the library. Both of these function first check the task's message queue to see if the requested message is there, and if the message was not found, they then use the run-time system's **test_predicate()** function, which simply checks to see whether a message has arrived but does not block to await one.

As part of the initialization of the library system, each task is assigned a set of ten (10) **pvm_message** structures that are available for sending and receiving. Each structure in the set is indexed so that the user is allowed to save (using **pvm_setsbuf()** and **pvm_setrbuf()**), recall (using **pvm_getsbuf()** and **pvm_getrbuf()**), and get information about (using **pvm_bufinfo()**) a message buffer. The set of **pvm_message** structures is allowed to grow (to more that 10) as needed.

3.4 Dynamic Process Group Support

PVM's support for process groups is also present in the library software. Support is provided for tasks to join and leave groups, as well as for the synchronization of tasks within a group and the broadcasting of data to tasks in a group. In order to perform the group operations, the group server process must be present and running.

The library's group server is implemented as a persistent Mentat object, and its definition is given in Figure 9e. (For a complete description of Mentat object types, see [7].) With the server being implemented as a Mentat object, it is no longer necessary for the programmer to explicitly start it before calling a group function. Instead, the server is started automatically when an application is started at the command line, and is destroyed when the "main" program calls **pvm_exit()** (the "main" program is considered to be the task that is started at the command line, not spawned ones). This means that there is one group server per application, and not one per virtual machine. When the "main" program spawns new tasks, the TID of the group server is passed to these new tasks. In this way, the children are able to access the same group server as their parent. The list of groups that the server maintains is initially set up for ten (10) groups, but more space can be reallocated as necessary.

Tasks are able to join (using **pvm_joingroup()**) and leave (**pvm_lvgroup()**) groups at will, without having to consult with any other tasks that are in the group. When a task joins a group, it is assigned an instance-number, which serves to identify the task in that group. (This is similar to the <*name*, *instance-number*> identification method of earlier versions of PVM.) Instance numbers start at 0 and go to the number of tasks in the group. If a task leaves a group and then rejoins it, there is no guarantee that it will receive the same instance-number. A task joining a group, however, will receive the lowest available instance-number. (The list of tasks in a group is actually an array (Figure 9a & 9b), so the instance number that is assigned is simply the index of the first free slot in the array.) Tasks can be members of different groups at the same time

Process synchronization (using the **pvm_barrier()** function) is accomplished by means of message passing. When a task calls **pvm_barrier()**, a message is sent to the group server notifying it of the barrier that is being waited for and the quorum that is to be met. The task then blocks and waits for a message from the group server telling it to proceed. When the server receives the barrier message, it first searches its list of barriers to see if the barrier already exists. If the barrier does not exist, the server creates a new **barrier_object** (Figure 9c) for this barrier, and adds it to its list of barriers. If the barrier

```
class task_object {
  public:
    char     tname[32]; // Task's name
    int      tid; // Task's TID.
    int      inumber; // Task's instance number.
};
```

(a)

```
class group_object {
  public:
    char           gname[32]; // Group name.
    int            gsize; // Group size.
    int            ttbl_len; // Number of tasks in the group.
    task_object    *tasks; // List of tasks in the group.
};
```

(b)

```
class object_node {
  public:
    task_object       t_obj;
    object_node       *next;
};
```

(c)

```
class barrier_object {
  public:
    char              bname[32]; // Barrier Name.
    int               quorum; // Number for quorum.
    int               num_waiting; // Number waiting currently.
    object_node       *waiting_objs; // List of waiting tasks.
    barrier_object    *next;
};
```

(d)

```
persistent mentat class group_server {
  private:
    group_object     *grp_tbl;   // Global group table.
    int              gtbl_len;   // Size of the table.
    barrier_object   *barriers;  // List of barriers.
    int              delete_barrier(barrier_object *);
    int              in_group(task_object, string *);
  public:
    int              initialize(void);
    int              cleanup(void);
    int              join_group(string *, task_object);
    int              leave_group(string *, task_object);
    int              getsize(string *);
    int              gettid(string *, int);
    int              barrier(task_object, string *, int);
    int              getinst(string *, int);
    task_object_list *get_tasks(string *);
};
```

(e)

Figure 9. Definition of the Group Server and Relevant Classes

exists, the server checks to see if the quorum for this barrier has been met. If the quorum has been met, the server sends a “go-ahead” message to each task that was blocked at this barrier, and removes the barrier from the list. The “go-ahead” messages are of a different type from the regular data messages. This is to ensure that a process that is blocked at a barrier would not proceed prematurely.

The broadcast function (**pvm_bcast()**) is implemented as a sequential send to each of the tasks in the group under consideration. The **pvm_bcast()** function first sends a message to the group server asking for a list of the tasks currently in the group. It then sends the data to each task in turn. This method is not the most efficient algorithm for implementing broadcasting, but it is the best that could be done without resorting to modifications of the Legion run-time system.

Also included in the library software are the **pvm_gather()**, **pvm_scatter()**, and **pvm_reduce()** functions. These are functions that perform global operations on the members of a group. The **pvm_scatter()** function distributes data to the members in a group, **pvm_gather()** collects data from the members of a group, and **pvm_reduce()** performs an arithmetic operation on the members of a group. The operations that are provided include global sums, products, minimums, and maximums (**PvmSum**, **PvmProduct**, **PvmMin**, and **PvmMax** respectively).

3.5 Multi-language Support

One of the features of PVM is its ability to support applications written in both the C and Fortran programming languages. The library software allows for the development of applications in either of the C, C++, or Fortran languages. Supporting C++ applications was straightforward since the Legion run-time system and Mentat applications are written in C++. Support for the other languages involved minor changes to the run-time system.

The major problem involved the calling of the constructors of global Legion system objects when the PVM application was written in either C or Fortran. The constructors for C++ objects are called in the linking phase of compilers, and since the application

would have to be compiled (at some point) by either a C or Fortran compiler, the linker function that calls system constructors would not be properly set. This meant that the library software would have to explicitly perform the functions of the constructors.

mentat_pred.h:

```
class pred_globals {
   private:
      ...
   public:
      void pred_globals();
      void initialize();
          ...
};
```

(a)

mentat_globals_init.c:

```
extern pred_globals pg;
... // Other mentat global objects.
void mentat_globals_init(void)
{
     pg.initialize();
     ... // Call other initialize functions.
};
```

(b)

pvm_utils.c:

```
void init_mentat_globals(void)
{
   mentat_globals_init();
}
```

(c)

Figure 10. Handling Global Constructors

Making sure that the global objects were initialized was accomplished in three steps:

1. A new function, **initialize()**, was added to each global Legion object class (Figure 10a). This function had the same signature as the class constructor and performed the same duties.
2. Create a new Legion system source file, **mentat_globals_init.c**, containing one function, **mentat_globals_init()**, which called the **initialize()** function of all the global objects from the original system source files (Figure 10b).

3. The library contains a function, **init_mentat_globals()**, which calls **mentat_globals_init()** (Figure 10c). **init_mentat_globals()** is called during the library's initialization phase.

This approach has one main advantage. If new global objects are added to the Legion system, the only changes that would be necessary would be changes to **mentat_globals_init.c**, and not to the library source files.This would increase the likelihood that the new global object would be properly initialized when the library initializes the other objects.

The Fortran library functions are implemented as calls to the corresponding C/C++ library functions. For example, a call to the **pvmfspawn()** function is actually a call to the **pvm_spawn()** function with the parameters organized accordingly. To implement the Fortran functions, the library makes use of the original PVM code and its use of the *m4* macro language processor to facilitate the proper passing of arguments between Fortran and C functions.

4.0 Performance of the Library Software

In order for the library to become widely used, it is necessary that it not introduce a large performance overhead when it is used and that it produce correct results. This section looks at the performance of PVM application programs when they are executed both in the PVM environment, and with the library software.

4.1 Timing and Bandwidth Measurements

This test used an application that is distributed with the PVM system. It consists of two (2) programs, a master that sends messages of varying sizes, and a worker program that acknowledges the receipt of the messages by sending a small message back to the master. The timings reported here (in Table 1) are the average of three (3) runs of the application, with each run being the average of twenty (20) sends and receives. These tests were conducted using PVM v3.0, on a network of eight (8) 40 MHz Sun Sparc 2's.

The results show that the library out-performs PVM for small messages (≤ 10,000 bytes). This is a result of the better message passing system of Mentat. The fact that the

Message Size (bytes)	PVM Time (msec)	PVM/Legion Time (msec.)	PVM Bandwidth (bytes/sec.)	PVM/Legion Bandwidth (bytes/sec)
100	8.81	8.48	11,310	11,800
1,000	9.87	9.75	101,260	102,550
10,000	27.52	26.58	363,380	376,220
100,000	194.23	196.78	514,860	509,880
1,000,000	1,920.62	1,951.62	520,670	512,790

Table 1. Timing and Bandwidth Measurements

library performs better than PVM for small messages is significant, because most applications send mainly small messages.

4.2 Benchmark Performance

The library was also tested with actual applications. In this case, the NAS benchmarks were used. These benchmark programs are a set of applications that "... embody the key computational and data-movement characteristics of typical processing in computational aerodynamics[6]." The benchmark programs consist of five kernels and three applications, and they are rigorous and come very close to "real" application. For testing the library, only the following kernels were used:

- Kernel EP: This kernel executes 2^{26} iteration os a loop in which a pair of random numbers are generated and tested for whether Gaussian random deviates can be made from them. This is one of the "embarrassingly parallel" kernels which is requires little or no communications between processors.
- Kernel MG: This kernel executes four iterations of the V-cycle multigrid algorithm to obtain an approximate solution to the discrete Poisson problem $\nabla^2 u = v$ on a 128x128x128 grid with periodic boundary conditions. This application utilizes both long- and short-distance communication.
- Kernel IS: This kernel performs 10 rankings of 2^{21} integer keys in the range [0, 19). This application uses frequent, but low-volume communications.

- Kernel CG: This kernel uses the power and conjugate gradient methods to approximate the smallest eigenvalue of a symmetric, positive definite, sparse matrix (of order 1,400) with a random pattern of non-zeros. (Memory restrictions prevented larger problem sizes from being attempted.) The communication patterns are long-distance and unstructured.
- Kernel FT: This kernel uses Fast Fourier Transforms on a 64x64x64 complex array to solve a 3-dimensional partial differential equation. (Memory restrictions prevented larger problem sizes from being attempted.) Communication patterns are structured and long-distance.

The performance of the benchmarks with the library system and with PVM v3.0 are shown in Table 2. As with the timing measurements, a network of eight (8) 40MHz Sun Sparc 2's was used (except for Kernel CG, which used 4[1]). In addition, all data was

	Benchmark Time (sec.)		Comm. Time (sec)		Comm. Vol. (MB)	
Application	PVM 3.0	PVM/ Legion	PVM 3.0	PVM/ Legion	PVM 3.0	PVM/ Legion
Kernel IS	226.28	256.28	202.61	207.00	140.02	140.02
Kernel EP	346.07	350.22	NA	NA	NA	NA
Kernel MG	123.76	110.06	61.94	59.37	48.77	48.77
Kernel CG	87.10	51.80	67.85	41.77	12.54	12.54
Kernel FT	92.63	87.56	70.84	65.43	49.00	49.00

Table 2. NAS Benchmarks Performance

sent using raw encoding and the **pvm_send()**/**pvm_recv()** functions.

1. Kernel CG requires a square number of processors, and 4 is the largest number of processors that was available in the testing environment.

5.0 Unsupported PVM Features

As much as it would have been desirable to support all the functionalities of PVM, the library software does lack some features. This section describes the functions that are not currently supported and explains those which are only partially supported.

- **pvm_addhosts()** & **pvm_delhosts()** :- these functions allow a user to dynamically add and delete hosts from the virtual machine. They will not supported due to the fact that the Legion design philosophy does not allow for a user to dynamically add/delete hosts. The Legion system handles these functions.

- **pvm_catchout()** :- this function causes the calling task to capture output from any tasks that it may later spawn. It will be supported when the necessary functionality is added to the run-time system.

- **pvm_config()** :- this function returns information concerning the current virtual machine configuration. The library's implementation bases its value of the relative CPU speed on the value of the "MIPS_RATING" variable from the Mentat configuration database file *config.db*. It is the user's responsibility to ensure that this value is accurate and reasonable. The default value is 1. (For more information on the *config.db* file, see [8].)

- **pvm_getopt()** & **pvm_setopt()** :- these functions retrieve and set various library options. Not all of the options to these functions are supported, however. The only supported options are ***PvmFragSize*** and ***PvmAutoErr,*** which control the size of the message fragments, and the printing of error messages respectively (the default values for these options are 4096 bytes and 1 respectively). Values for the tracing options as well as the routing options may be passed to the functions, but they are currently disregarded. Passing values to the ***PvmDebugMask***, ***PvmOutputTid***, ***PvmOutputCode***, and ***PvmResvTids*** options is reported as a warning. (These options control the debugging mask, the output device for a task's children, the message tag to be used for children's output messages, and the reserving of message tags and TIDS respectively.)

- **pvm_halt()** :- this function is only partially supported. The PVM function shuts down the entire PVM system (including remote tasks and daemons), while the library's implementation simply destroys any running tasks and the group server. The underlying Legion system is left running to be shut down in its normal manner.

- **pvm_hostsync()** :- this function gets the time-of-day from a remote task. Currently it is not supported, but it may be when signal support is added to Legion.

- **pvm_initsend()** :- the ***PvmDataInPlace*** option to this function is not fully supported. Although it may be used, the data is actually packed as though the ***PvmDataDefault*** option was used. (The ***PvmDataInPlace*** option specifies that data is to be place during packing. The message buffer contains only the sizes of, and pointers to, the data items to be sent.)

- **pvm_insert() & pvm_lookup()** :- these functions are used to insert data into, and retrieve data from, the pvmd database. (The pvmd database is used to store information such as TID's and make them available to tasks within the virtual machine. This is useful for applications like the group-server.) The library does not make use of such a database, and thus these functions are not supported.

- **pvm_notify()** & **pvm_sendsig()** :- **pvm_notify()** tells a task when an event of interest, for example a host going down, happens. **pvm_sendsig()** sends a signal to a specific task. These functions are not currently supported, but could be when signal support is added to Legion.

- **pvm_reg_hoster()** & **pvm_reg_tasker()** :- **pvm_reg_hoster()** registers the calling task as being responsible for adding new hosts to the virtual machine, while **pvm_reg_tasker()** registers the calling task as being responsible for starting new tasks (the PVM daemons usually handle these tasks). These functions will not be supported due to the fact that Legion does not allow for users to add hosts "on the fly" or for user defined objects responsible for starting new tasks.

- **pvm_reg_rm()** :- pvm_reg_rm() registers the calling task as a resource manager, meaning that it will play a role in scheduling policies. This function will not be supported since the Legion daemons already handle scheduling, and Legion does not allow for users to take over this responsibility.

- **pvm_spawn()** :- the ***PvmTaskTrace***, ***PvmTaskDebug***, and ***PvmMppFront*** options to this function are not currently supported. The ***PvmTaskTrace*** option implies that the spawned tasks will generate tracing data, the ***PvmTaskDebug*** option causes the spawned tasks to start up under a debugger, and the ***PvmMppFront*** option causes the spawned tasks to start on an MPP front-end. In addition, since Legion does not currently provide for the passing of arguments to tasks created by the IM, the *argv* parameter to **pvm_spawn()** is disregarded.

- **pvm_start_pvmd()** :- this function allows the user to start a new PVM daemon on another host. This function is used mainly when adding new hosts to the virtual machine. Since this ability (to add new hosts) is not provided to the Legion user, this function will not be supported.

- **pvm_tasks()** :- this function returns information about the tasks that are currently running on the virtual machine. In the library's implementation, the fields of the pvmtaskinfo structures that are returned, have the following values:

  `ti_tid` :- the TID of the task is in this field.

  `ti_ptid` :- this field is supposed to contain the TID of the task's parent. However, the library is unable to determine the parent of an arbitrary task (without explicitly asking the task). Consequently, this field contains the TID of the IM on the host on which the task resides. (Legion considers all objects to be children of the IM's, so technically this information is correct.)

  `ti_host` :- this field is supposed to contain the TID of the daemon on the host that the task is on. The library returns the TID of the IM on the host on which the task resides. (Same as for ti_ptid.)

`ti_flags` :- this field is supposed to indicate the status of the task: is it connected to the daemon, or is it waiting for authorization, is its connection being closed, or is it a special (hoster, resource-manager, or tasker) task. The library returns a value of 0 in this field since none of the PVM task states have any meaning in the Legion environment.

`ti_a_out` :- this field contains the name of the task. The library returns the correct name for all tasks except the one that was started by hand on the command line. For that task, the library returns the string ***PVM_parent***.

`ti_pid` :- this field contains the process ID of the task.

Although some of the more advanced features of PVM are missing from this version of the library software, it should be noted that all the important features are included and that most “real” applications could be developed with the features that are included.

6.0 Building and Using the Library

6.1 Obtaining the Software

The library software can be obtained in the same manner as the Mentat system software. Complete information on getting access to the Mentat distribution files can be gotten by sending mail to mentat@Virginia.edu.

6.2 Building the Library

The library is distributed in compressed tar format. It is recommended that the distribution file be copied to home directory of the Mentat account that was set up when the Mentat system was installed. To unpack the source one must enter the following commands:

```
$  uncompress pvmlegion.tar.Z
$  tar xvf pvmlegion.tar
```

These commands will create a directory, *pvmlegion*, and place the source code for the library there. This directory should contain eleven (11) source (and header) files, as well as three subdirectories, the PVM Applications Compiler (***pac***), and a makefile. One subdirectory will contain macros that are used during the compilation of the library on the

different platforms, another contains the source code for the Fortran library functions, and the final one contains example applications. There should also be a README file that further explains the contents of the subdirectories.

To build the library, change to the *pvmlegion* directory and enter the command "make all". The makefile will determine which platform it is currently on and build ***libpvm3.a libfpvm3.a*** and ***group_server*** (the group server executable) accordingly (the steps in the compilation process are different for the different platforms). During compilation, the compiler will output a number of error messages of the form "**error!: Illegal use of Mentat class member variable!**" or of the form "**error!: Identifier *name* is being redeclared**". These messages can be safely disregarded. The warnings concerning unreachable statements and unset variables can also be disregarded.

When the compilation is complete, the libraries will be placed in the proper subdirectory of the $MENTAT_LIB directory. (For example, on a SUN4 machine the library will be placed in $MENTAT_LIB/att. It is assumed that Mentat has been already installed and the Mentat environment variables are properly set.) The group server executable will be placed in $MENTAT_USR_BIN. The user should copy ***pac*** to $MENTAT_BIN.

6.3 Compiling Applications with the Library

The PVM Applications Compiler (**pac**) is used to compile PVM applications that need the library. This compiler is actually a shell program that first compiles the application into a "**.o**" object file using the appropriate language compiler, and then uses a C++ compiler to link the object file and the necessary libraries to produce an executable. Table 3 lists the platforms and compilers with which the library has been tested.

The syntax for the **pac** command is:

pac [**-C**] [**-C++**] [**-F**] [**-v**] [*Compiler options*] *sourcefile* [*objectfiles*]

where:

-C
Specifies that the PVM application is written in C.

-C++
Specifies that the PVM applications is written in C++.

-F

Platforms	C Compiler	C++ Compiler	Fortran Compiler
SUN4†	cc	CC_CenterLine	f77
SOLARIS	cc	CC_CenterLine	f77
SGI	cc	CC	f77
RS6000	gcc	g++	xlf

† Requires SunOS 4.1.2 (at least)

Table 3. Supported Platforms and Compilers

Specifies that the PVM applications is written in Fortran.

-v
Verbose mode: show explicitly what **pac** is doing.

Compiler options
Other Fortran, C, or C++ compiler options that may be necessary.

sourcefile
The name of the PVM application source file.

objectfile
The names of any other needed object files.

For example:

```
pac -F -v test.f -o test
pac -C -DDEBUG -g hello.c -o hello
```

On-line help for ***pac*** is also available through man pages (type: `man` ***pac*** or `man` ***pvm-legion***).

A new error code has also been added to *pvm3.h*. As a result, the user should either use the header files that are supplied with the library or make the necessary changes to their header file. The change to be made involves adding the following line to the *pvm3.h* header file:

```
#define PvmNotSupported -34 /* A function or option is not yet sup-
                               ported. */
```

6.4 System Dependent Libraries

In order for **pac** to successfully compile Fortran PVM applications, it must make use of certain libraries that may be in different locations on different systems. The following libraries are necessary, and the user must ensure that **pac** can reference them:

Platforms	Libraries
SUN4	libF77.a
SOLARIS	libF77.a, libM77.a, libsunmath.a
SGI	libftn.so, libbsd.a, libl.a
RS6000	libbsd.a, libxlf90.a, libl.a

Table 4. System Dependent Libraries

In order to specify the path to these libraries, the user must change the value of the FLDIRS variable in the source code for **pac**. The value of the variable is set based on the platform that is being used.

7.0 References

[1] Vaidy S. Sunderam, *PVM: A Framework for Parallel Computing*. Concurrency: Practice and Experience, Vol. 2, No. 4, December 1990, pp315-339.

[2] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[3] Andrew S. Grimshaw, *Easy to Use Object-Oriented Parallel Programming with Mentat*. IEEE Computer, May 1993, pp39-51.

[4] Andrew S. Grimshaw, Jon B. Weissman, W. Timothy Strayer, *Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing.* Computer Science Technical Report CS-93-40, University of Virginia, July 1993.

[5] G. A. Geist, V. S. Sunderam, *Network Based Concurrent Computing on the PVM System*. Journal of Concurrency: Practice and Experience, Vol. 4, No. 4, June 1992, pp293-311.

[6] S. White, A. Ålund, and V. S. Sunderam, *Performance of the NAS Parallel Benchmarks on PVM Based Networks*. RNR Technical Report RNR-94-008, NASA Ames Research Center, May 1994.

[7] The Mentat Research Group, *Mentat 2.8 Programming Language Reference Manual*, 1995.

[8] The Mentat Research Group, *Mentat 2.8 Users Manual*, 1995.

[9] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, Paul F. Reynolds Jr., *A Synopsis of the Legion Project*. Computer Science Technical Report CS-94-20, University of Virginia, June 1994.

[10] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, Paul F. Reynolds Jr., *Legion: The Next Logical Step Towards a Nationwide Virtual Computer.* Computer Science Technical Report CS-94-31, University of Virginia, June 1994.

[11] Andrew S. Grimshaw, Anh Nguyen-Tuong, William A. Wulf, *Campus-Wide Computing: Early Results Using Legion at the University of Virginia*. Computer Science Technical Report CS-95-19, University of Virginia, March 1995.