Beyond PVM 3.4: What We've Learned, What's Next, and Why

G.A. Geist, J.A. Kohl, P.M. Papadopoulos, and S.L. Scott

Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN, 37831-6367, USA

Abstract. This paper explores the foundations and philosophy that have made PVM both effective and widespread: a simple system abstraction and API, transparent heterogeneity, and dynamic system configuration. From a high-performance programming point of view, we examine the features that make PVM useful and those that make hardware-level performance difficult to achieve. The key conclusion from this analysis is that PVM, MPI, and similar paradigms suffer from a monolithic approach to the distributed computing problem. The approaches simply cannot meet the large range of service requirements for advanced distributed computing environments. The notion of a Generalized Plug-In Machine (GPM) is introduced that allows programs to exert better control over their operating environment. This environment has the potential to provide mechanisms for better performance, richer system dynamics, and fault-tolerance. Pluggable components, such as messaging substrates, dynamically-attached debugging agents, or complete virtual machines that can be joined together, form an operating environment that can be customized on-the-fly. Generalizations of current PVM plug-ins (resource managers, hosters, taskers) that lead to this next-generation environment are discussed, and inherent challenges, such as eliminating the master PVM daemon and providing the pluggable substrate, are examined.

1 Introduction

PVM is a successful research project that started with a simple idea: using commodity networks, workstations can be managed together and programmed as a single virtual resource. The basic PVM system has gone through three major revisions, with each new release using fundamentally different internal designs. There are indeed three key principles that have guided the design and implementation: a simple API, transparent heterogeneity, and dynamic system configuration. The simple API allows messaging, virtual machine control, task control, event notification, event handlers, and a message mailbox all to be accessed and controlled using only about 60 user-level library routines. Transparent heterogeneity makes it easy to construct programs that interoperate across different machine architectures, networks, programming languages, and operating systems. Dynamics allow the virtual machine configuration to change and the number of tasks that make up a parallel/distributed computation to grow and shrink under program control. Proponents of PVM have exploited these features and learned to live within the boundaries that the system provides. However, these boundaries have often translated to limited messaging semantics and difficulties in achieving peak performance on high-bandwidth networks. This paper lays the groundwork for where we believe distributed computing should lead in the long term: a dynamically configurable system that allows programs to define their performance/feature tradeoffs to optimize and customize the environment.

We see a common theme in all popular distributed computing paradigms including PVM: each mandates a particular programming style and builds a monolithic operating environment into which user programs must "fit. " MPI-1, for example, mandates an SPMD-style static process model with no job control. This maximizes performance by minimizing dynamics and works very well in static environments. Programs that fit into the MPI system are well served by its speed and rich messaging semantics. PVM, on the other hand, allows programs to dynamically change the number of tasks and add or subtract resources. However, programs in general pay a performance penalty for this flexibility. Even though MPI and PVM provide very useful environments, some programs simply are not able to find the right mix of tools or are paying for unwanted functionality. Here, the monolithic approach breaks down and a more flexible pluggable substrate is needed. This idea is certainly not a unique and has been successfully applied in other areas – the Mach operating system is built on the microkernel approach, Linux has pluggable modules to extend functionality, and Horus uses a "Lego Block" analogy to build custom network protocols. By extending and generalizing these ideas to parallel/distributed computing, programs will be able to customize their operating environment to achieve their own custom performance/functionality tradeoffs.

PVM and MPI both exhibit desirable features and APIs that should and will transfer to the next generation. Our next generation environment will focus on distributed resources and methods to bring better control and performance to programs, rather than attempting to rewrite messaging semantics. However, a significant new class of applications will be enabled if the runtime system is logically decoupled from the semantics of the user interface and new dynamic interactions are defined. The challenge is to implement a pluggable substrate that is simultaneously efficient, dynamic, and robust. The next generation environment will support virtual machines that can roam across networks to attach and detach resources as program requirements change. We see this as a fundamental enabling technology for completely new classes of applications. PVM already defines three plug-in interfaces that enable tasks to modify default policies and mechanisms: starting hosts, starting tasks, or managing resources. However, if the programmer wants to change the underlying message substrate, needs message encryption, or requires different event notification properties, a truly Herculean effort is required to modify the PVM runtime system. In addition, if two virtual machines are separately instantiated, they can never be merged together (or a single machine split into two). This paper explores the concept of pluggability at three layers – low level messaging, mid-level resource and analysis tools, and high-level control of the virtual machine that encompasses merging, splitting, and migration.

This paper will reflect upon the properties of PVM 3.4 in relation to these three levels, how these have influenced the internal design design of PVM, and how they provide a stepping stone to a more dynamic, pluggable environment.

2 The Basic PVM Design Assumptions

PVM was designed for heterogeneous interoperation across TCP/IP networks. One should refer to [1] for complete details of the protocols and inner workings of PVM 3.x. The API has three classes of functions: messaging, task control, and machine control.

2.1 Messaging Assumptions and their Relation to Internal Design

The fundamental messaging unit internal to PVM is a message *fragment*, with a user-level message being made up of one or more fragments. Fragments are usually compatible with the TCP/UDP limits set at run time, but may take on any length. The default operational mode (which guarantees interoperability across machines) requires a task first to completely fragment the message using `pvm_pack` before sending the message to its destination. This serialization (fragmentation of an entire message followed by a send) introduces a message copy, increases latency, and decreases bandwidth. Different message semantics, like those found in MPI, allow the underlying system to pipeline the pack and send operations so that long messages may overlap the sending of one fragment with the packing of a subsequent one. Packing does allow translation of the data to its XDR format, a costly but often necessary step. It also provides a means to conglomerate many smaller data chunks into a single buffer for network transfer (which greatly improves efficiency on traditional networks). The internal design of PVM reflects these pack/send semantics and makes it quite challenging to retrofit the code to use pipelining and other performance improvements. PVM messaging also assumes synchronous messaging – a data buffer can be re-used as soon as `pvm_send` returns. Similar to packing, the internal implementation makes asynchronous messages (or post/test-for-completion operations) difficult to include. The current PVM API has influenced the internal design to the point where it is difficult to map other semantics efficiently on top of the existing system. The lesson learned here is that the semantics of a particular messaging style (ala' PVM or MPI) must be decoupled from the low-level infrastructure. This does not imply that the PVM (or MPI) interface should be thrown away, but rather the internal implementation needs to be more flexible to handle a wide variety communication semantics like: one-sided operations (puts and gets), asynchronous messaging, encryption (per message or per link), streams (for video and audio), quality of service links, and linked streams.

2.2 Plug in Communication Modules

As distributed computing has developed, it has become clear that no one monolithic system can handle efficiently all the desired communication styles. Extensibility of a core system is essential to achieve critical performance and gives a practical method to manage multiple communication styles. Because messaging is extremely important to system performance and evaluation, the lowest layers must be able to be swapped out for different scenarios. For example, send/receive message passing is quite different from one-sided communication semantics. Low-level performance can be significantly affected if support for both is automatically installed when not needed by an application. The inefficiency comes from the fact that incoming messages need to be checked among different communication methods to determine the correct handling sequence. If a particular message style (e.g. a put) is never used, then a reduction in overhead can be made by eliminating this as a checked-for style. On MPPs, for example, is it unnecessary to fragment messages or provide reliable transmission because it is usually guaranteed by the message system. On the other hand, communicating over a WAN requires fragmentation, timeouts/retries, and connection failure monitoring. A user should be able to write a distributed application and have the *runtime* system select which method(s) are needed for the particular run. The key to success will be to design plug-in communication stacks (similar to those found in Horus [10]) that can be traversed quickly and efficiently. To get optimum performance, it may require the user to use strongly-typed messaging like MPI. However, runtime pluggability can still give significant advantages without requiring users to dramatically change code. For example, one may desire to encrypt certain communication links only if the virtual machine spans several domains. Runtime pluggability would allow an encrypted link to be installed without user code modification. The next generation pluggable machine will have to strike a better balance among performance (or the ability to optimize performance), flexibility, and interoperability. Due to the large body of research on communication methods, this lowest level of pluggability is probably the most straightforward goal to achieve.

3 Middle-Level Tools: Current Design, Features, Needs

The next-generation pluggable computing environment will provide a fertile platform for a variety of "middle-level" tools that manipulate and administer system resources. Fundamentally different from traditional virtual systems, each user application will customize its own computing environment interactively, to bring the desired computational and communication resources together as needed over the course of its execution. Existing systems have already taken some steps toward this level of flexibility, but there are many issues yet to be resolved before a truly pluggable system can be achieved. This section overviews the current pluggable computing tools and their features, and explores future needs along with the accompanying obstacles.

The concept of "pluggable" interfaces is not a new one, but rather stems from the fundamental software engineering principles of modularity and encapsulation of function. If a system is designed carefully by separating concerns and generalizing the handling of similar operations, then it is straightforward to "swap out" one implementation detail for another. Much of object-oriented methodology is built on this premise. If an "object" has a well-defined interface and encapsulates its internal state and operations, then a new object, with a compatible interface but different internal implementation, can be inserted in place of the first object without disturbing the overall functioning of the system.

PVM has supported this type of modularity since PVM 3.3, with the existence of "hoster," "tasker," and "resource manager (RM)" tools that can be instantiated to take over handling of certain system functions. A hoster program can be started to customize the procedure for adding new hosts to the virtual machine. Hence, it currently is possible, but not simple for a user to provide customized authenticated startup of remote PVM daemons (PVMDs). A tasker can be started on each host of the virtual machine to handle the spawning of tasks there, e.g. to allow debuggers to intervene in a task's execution. Installing an RM in the virtual machine is equivalent to a complete "take-over" of all scheduling policies, including host adding and deleting, various event notifications, and task information and spawning placement. PVM 3.4 also introduces the concept of a "tracer" for automatically collecting trace event messages from internal library instrumentation, to alleviate the need for a full tracing tool such as XPVM [4, 5] when collecting traces off line.

In all of the above cases, the tool "plugs itself in" by registering with a PVMD to handle certain classes of requests. Subsequently, when one of the desired requests arrives from a user application, the PVMD passes control to the registered tool to handle that request. In this sense, certain aspects of the PVM system resources and scheduling can be customized on-the-fly by "plugging in" user replacements for the default handlers. Unfortunately, like most traditional computing environments, PVM did not originally support this type of interactive customization by design, and had to be specifically re-coded to allow alternate external handling of the desired operations.

Netscape's Navigator web browser, version 3.0 [9], supports a similar type of pluggable interface. While the browser is running, a "plug-in" can be registered to handle certain types of web page contents. When one of these content types occurs, the browser automatically forwards the data to a plug-in task or thread which then processes the data and either displays the results or returns to the browser for inline display. Using the explicit Mime typing information carried by all web pages, data of any type can be handled in a customized manner.

With both PVM and the Netscape Navigator, the pluggability is of a static nature in that, even though the plug-in tools can be added at run time, there is a fixed and limited protocol within which the plug-ins must operate. The plug-ins cannot transcend their predefined range of functionality. Therefore, for the next-generation pluggable system to be fully general, the pluggable architecture must also encompass the very protocols used by the plug-in elements to interact with the system. The system must be built with very little inherent functionality, primarily just a simple interpreter for "self-defining" protocols that allows all other aspects of the system to be generically connected as needed. Even the default system handlers will use the same protocol interface to attach to the "micro-kernel," which essentially serves only as a point of contact.

An important issue surrounding this flexibility is the potential trade-off for performance. Because the plug-in protocols are generalized and require interpretation, the computation required to process them is increased and the overall performance of the system is degraded. While this may be acceptable for certain types of plug-in tools, such as debuggers, it could be disastrous for tools which require greater bandwidth, such as tracing or multimedia tools. One solution is to optimize the use of self-defining protocols. PVM 3.4 already applies an optimization to its generalized tracing protocol by only sending trace event descriptors with the first occurrence of each trace event, thereby reducing needed bandwidth and tracing overhead [5]. Alternatively, a better solution may be to provide a few static "high-performance" protocols for certain well-defined interactions. These special-purpose protocols, along with the generalized protocols, would result in a more balanced pluggable system, with both ample flexibility and respectable performance [1].

When the pluggable approach is applied to a concurrent application with many tasks executing in parallel, several other issues arise. Suppose that the user wishes to interactively attach a tool to a set of running application tasks, as is done with the CUMULVS system for interactive visualization and computational steering [2]. Currently, for CUMULVS to be able to connect to an application the user must instrument the application so that each task periodically passes control to a CUMULVS library routine. Within this routine the CUMULVS protocols are processed to construct connections with CUMULVS viewer programs and to exchange information and control directives. Because the user application tasks may not be tightly synchronized, race conditions exist wherein the application could incorrectly receive messages that are part of the CUMULVS protocols. To prevent this from happening the user must set aside a small portion of the message "tag space" to prevent interception of CUMULVS-tagged messages by the application, and vice versa.

Progress has already been made regarding these issues in the latest release of PVM 3.4 [3]. The introduction of "message handlers" now allows protocols, like that required for CUMULVS, to be handled transparently. Instead of passing control to some explicit protocol interpreter, message handlers can be defined to intercept any incoming protocol messages automatically, and process the protocol interactions to update the necessary state information. Because these message handlers are invoked internally without any user intervention, it is especially critical to guarantee that message tag spaces do not overlap. To simplify this problem PVM 3.4 uses the concept of "context" to encapsulate the different realms of communication within distinct tag spaces. Aside from a message tag, each message is identified with a system-generated context. A task can only send

[1] This is, in fact, the philosophy under which the PVM system was developed.

and receive messages within its currently selected context, and likewise message handlers are defined to process messages within a given context. This alleviates any user specification of tag spaces for avoiding incorrect message delivery.

Though the above strides have been made toward more flexible pluggable tool architectures, more fundamental changes are necessary to fully realize the potential of such systems. The integration of generalized or self-defining protocols is required. However, performance is important and cannot be overlooked entirely in favor of flexibility.

4 Virtual Machine Control: Splitting, Merging, Migration

4.1 The Role of the Master PVM Daemon

A user's virtual machine is made up of a user-defined set of communicating daemon processes called PVMDs. To expand the virtual machine, a new daemon is started on the new host and configuration parameters are exchanged to reflect the new configuration. The PVM design revolves around a master daemon that keeps and distributes the current configuration. Furthermore, new hosts are added to the current virtual machine by a protocol that must go through the master. If the master fails, then all non-master daemons shut themselves down and the entire virtual machine collapses. Because only a single master keeps and modifies the critical configuration information (the host table), several race conditions are automatically eliminated. This leads to a more stable and predictable system for most users. There are, however, three significant consequences of this implementation: virtual machines started independently of one another may not join into a single virtual machine; a single virtual machine may not split to form multiple virtual machines; and virtual machines cannot "move" and may only grow and shrink while rooted at a single master. One could elect multiple masters to eliminate the single point of failure, but this requires that the pool of masters closely coordinate the addition and deletion of hosts. New daemons could be promoted to this master pool if an old master fails. However, to achieve coherency of configuration changes, the addition, deletion and election protocols must themselves be fault-tolerant. While not unsolvable, this represents a significant increase in complexity in handling the configuration of the current virtual machine. It is indeed a challenging problem and will require considerable effort to examine the tradeoffs of different solution scenarios.

4.2 Naming and Migrating Tasks

PVM task ids (or tids) are named relative to a host which in turn is named relative to the master PVMD. Therefore, encoded in the tid, which is a 32 bit integer, is the physical host of a particular process. If one desires a PVM task to migrate, then a new tid must be assigned. Re-assigning the tid so that it is consistently reflected across all tasks in the virtual machine becomes very complicated due to the large number of race conditions that must be solved. Systems like Condor [7] have wrestled with the task naming problem as they move tasks around. One logical solution is to virtualize task ids so that a virtual-to-real-space mapping is made at every call. This can be implemented efficiently in a similar manner to virtual memory. However, the updating of virtual-to-real-tid maps across an entire virtual machine may be problematic because of host failures during the update process. Migration of tasks also presents a problem when tasks have open files. The file may be unavailable on the new system, or in the case of Unix operating systems, the file may itself be an abstraction of some other physical entity (e.g. a communication socket). It is clear that migrating tasks and migrating or merging virtual machines present very challenging problems with complex solutions. A large part of where PVM is going next will revolve around these two important issues.

4.3 New Capabilities of Dynamic Machines

The capability of dynamically merging and splitting virtual machines has the advantage of allowing users to generate roaming applications. While the pluggability of low-level communication protocols and middle-level tools provide practical engineering benefit, roaming opens up whole new applications classes. For example, roaming applications may discover better resources and migrate their virtual machine (and associated tasks) to a more desirable configuration. While PVM provides some capability to migrate application components or even complete applications, it is not easy or straightforward. By making migration a pluggable part of the base functionality (included only when wanted by the program), better resource utilization can be achieved. Furthermore, multiphase programs can more easily be created where the environment adjusts to changing resource needs.

A further benefit of dynamic merging and splitting is that portions of applications can create their own virtual machine to resolve specific sub-problem components and then briefly join with other virtual machines to continue processing on the larger problem. Modeling of complex real-world problems could greatly benefit from this capability. A simple example is to use separate application suites to model the various individual airframe components for an aircraft, with brief application joins to introduce the various interacting boundary conditions. Finally, fault tolerance of an application can be easily achieved more easily when running in an environment where the masterless virtual machine supports migration.

5 PVM 3.4 A first step towards GPM

The new features included in PVM 3.4 provide a stepping stone to the next generation distributed computing environment. As such, PVM 3.4 will allow users to develop much more flexible, dynamic, and fault tolerant applications. In addition, feedback from users of PVM 3.4 will help guide the development of the environment.

While the actual number of new functions in PVM 3.4 is small, these functions provide the biggest leap in PVM capabilities since PVM 3.0 came out in 1993. The functions provide communication context, message handlers, and a tuple space called message box.

The ability to send messages in different communication contexts is a fundamental requirement for parallel tools and applications that must interact with each other. It is also a requirement for the development of safe parallel libraries. Context, which PVM defines as a unique system-created tag, s sent with each message. The matching receive function must match the context, destination, and message tag fields for the message to be received. (wild cards are allowed for destination and message tag but not for context). In the past, PVM applications had to divide up the message tag space to mimic context capabilities. With PVM 3.4 there are built in functions to create, set, and free context values.

By defining the context to be system-wide unique, PVM continues to allow the dynamic generation and destruction of tasks. And by defining that all PVM tasks have a *base* context by default, all existing PVM applications continue to work unchanged. The combination of these features allows parallel tools developers to create visualization and monitoring packages that can attach to existing PVM applications, extract information, and detach without concern about interfering with the application.

GPM's ability to dynamically plug-in middle-layer tools and applications is predicated on the existence of a similar if not identical communication context paradigm to PVM 3.4.

PVM has always had message handlers internally, which were used for controlling the virtual machine. In PVM 3.4 the ability to define and delete message handlers has been raised up to the user level. To add a message handler, an application task calls

```
handler_id = pvm_addmhf( src, tag, context, function );
```

Thereafter, whenever a message arrives at this task with the specified source, message tag, and communication context, the specified function is executed. The function is passed the an ID so that it may unpack the message if required. PVM 3.4 places no restrictions on the complexity of the function. It is free to make system calls or other PVM calls.

With the functionality provided by pvm_addmhf() it is possible to build one-sided communication, active messages, applications that trigger other applications on certain events, fault recovery tools and schedulers, and so on. For example, instead of an error inside an application printing an error message, the event could be made to invoke a parallel debugger focused on the area of the problem. Another example would be a distributed data mining application that finds an interesting correlation and triggers a response in all the associated searching tasks. The existence of pvm_addmhf() allows tasks within an application to dynamically adapt and take on new functionality whenever a message handler is invoked.

In GPM this ability to dynamically add new functionality will have to be extended to include the underlying system as well as the user tasks. One could envision a message handler defined inside the virtual machine daemons that when triggered by the application would spawn off intelligent agents to seek out the requested software module from Web repositories. These trusted "children" agents could retrieve the module and use another message handler to cause the daemon to load the module, incorporating its new features.

In a typical message passing system, messages are transitive and the focus is often on making their existence as short as possible, i.e. decrease latency and increase bandwidth. There are many situations in distributed applications seen today in which programming would be much easier if there was a way to have persistent messages. This is the purpose of the new *Message Box* feature in PVM 3.4. The Message Box is an internal tuple space in the virtual machine. Tasks can use regular PVM pack routines to create an arbitrary message and then use *pvm_putinfo()* to place this message into the message box with an associated name. Copies of this message can be retrieved by any PVM task that knows the name. And if the name is unknown or changing dynamically, then *pvm_getmboxinfo()* can be used to find the list of names active in the Message Box.

The four functions that make up the Message Box in PVM 3.4 are:

```
index = pvm_putinfo( name, msgbuf, flag )
    pvm_recvinfo( name, index, flag )
    pvm_delinfo( name, index, flag )
    pvm_getmboxinfo( pattern, names[], struct info[] )
```

The `flag` defines the properties of the stored message, such as, who is allowed to delete this message, does this name allow multiple instances of messages, does a *put* overwrite the message? The `flag` argument also allows extension of this interface as PVM 3.4 users give us feedback on how they use the features of message boxes.

While the tuple space could be used as a distributed shared memory, similar to the Linda [6] system, the granularity of the PVM 3.4 implementation is better suited to large grained data storage.

Here are just a few of the many potential Message Box uses. A visualization tool spontaneously comes to life and finds out where and how to connect to a large distributed simulation. A scheduling tool retrieves information left by a resource monitor. A new team member learns how to connect to an ongoing collaboration. A debugging tool retrieves a message left by a performance monitor that indicates which of the thousands of tasks is most likely a bottleneck. Many of these capabilities are directly applicable to the GPM environment, and some method to have persistent messages will be a part of the GPM design.

The addition of communication contexts, message handlers, and message boxes to the parallel virtual machine environment allows developers to take a big leap forward in the capabilities of their distributed applications. PVM 3.4 is a useful tool for the development of much more dynamic, fault tolerant distributed applications, but as illustrated above, it is also a testbed for software and features that will make up the next generation heterogeneous distributed computing environment.

6 Conclusions

This paper has presented the basic concepts for a pluggable virtual machine. The pluggability was described at three levels: low level communication, middle-level tools, and virtual machine merging, splitting, and migration. The design and philosophy of what is next were motivated by lessons learned over the past few years with the PVM environment. The latest release of PVM version 3.4 and its new features like context, message handlers and message boxes are seen as the first key steps to building the generalized pluggable machine. Clearly, the highest technical hurdle to jump is to enable efficient migration of tasks and virtual machines by handling task renaming and being able to consistently represent the configuration of the virtual machine. Pluggability is being explored to improve performance, provide greater flexibility, and open up new classes of roaming applications.

References

1. A. Beguelin, J. Dongarra, G. A. Geist, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.
2. G. A. Geist, J. A. Kohl, P. M. Papadopoulos, "CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications," SIAM, August 1996.
3. G. A. Geist II, J. A. Kohl, R. Manchek, P. M. Papadopoulos, "New Features of PVM 3.4, " 1995 EuroPVM User's Group Meeting, Lyon, France, September 1995.
4. J. A. Kohl, G. A. Geist, "XPVM 1.0 User's Guide," Technical Report ORNL/TM-12981, Oak Ridge National Laboratory, Oak Ridge, TN, April, 1995.
5. J. A. Kohl, G. A. Geist, "The PVM 3.4 Tracing Facility and XPVM 1.1," Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS-29), Heterogeneous Processing Minitrack in the Software Technology Track, Maui, Hawaii, January 3-6, 1996.
6. N. Carriero and D. Gelernter. "Linda and Message Passing: What Have We Learned?" Technical Report 984, Yale University Department of Computer Science, Sept. 1993.
7. J. Pruyne and M. Livny, "Interfacing Condor and PVM to Harness the Cycles of Workstation Clusters", Journal on Future Generations of Computer Systems, Vol. 12, 1996
8. Message Passing Interface Forum. MPI: A message-passing interface standard. "International Journal of Supercomputer Applications and High Performance Computing", International Journal of Supercomputer Applications and High Performance Computing, Volume 8, Number 3/4, 1994.
9. Netscape Navigator 3.0, Netscape Communications Corporation.
10. R. Van Renesse, K. P. Birman, S. Maffeis, "Horus, a Flexible Group Communication System", Comm. ACM, April, 1996.

This article was processed using the LaTeX macro package with LLNCS style