Client Server Computing on Message Passing Systems: Experiences with PVM-RPC *

A. T. Krantz and V. S. Sunderam

Dept. of Math & CS, Emory University, Atlanta GA 30322, USA
email: {atk, vss}@mathcs.emory.edu

Abstract. The relationship between client-server distributed computing and message-passing parallel processing is explored in this work through an experimental RPC framework for the PVM system. The project investigates the potential for RPC to complement asynchronous message passing in PVM – both to expand the domain of applications, and to evaluate the effectiveness of client-server computing for traditional scientific and numerical problem solving. Our design is intuitive and straightforward, and enables servers and clients to be developed with a minimum of additional effort in terms of programming and logistics. Our experiences with early implementations of PVM-RPC indicate that the client-server model is reasonably compatible with message passing and scientific algorithms, and furthermore, that the RPC layer introduces little if any performance overhead into message passing systems. Furthermore, support for simple versions of useful features such as transparent failure resilience and automatic load balancing are facilitated in this model.

1 Introduction

Parallel Virtual Machine (PVM)[7] is a software system for heterogeneous parallel and distributed computing on networked systems. PVM, which is based on the message passing model, has become a de-facto standard and a widely used system due to its design effectiveness, implementation portability, and robustness. However, providing only a message-passing interface has restricted PVM application categories significantly. Furthermore, the message-passing model requires a number of potentially complex tasks to be programmed explicitly by the user, including process identification and table maintenance; message preparation, transmission/reception, ordering, and discrimination; and task synchronization.

In contrast, traditional distributed computing is based on the "client-server" model. In this paradigm, entities known as servers provide services that may be utilized by clients. Implicit in this abstract definition is the potential for varying levels of implementation semantics. Typical implementations of systems include

* Research supported by Army Research Office grant DAAH04-96-1-0083 U. S. Department of Energy Grant No. DE-FG05-91ER25105, and the National Science Foundation Award Nos. CCR-9523544 and ASC-9527186.

location transparency, data-representation, heterogeneity, failure resilience, and dynamic replication of servers to meet increased load. A popular manifestation of client-server computing is the the Remote Procedure Call (RPC)[8, 3] interface, that avoids many of the pitfalls of asynchronous message-passing programming. For this and other reasons, including those mentioned above, the overwhelming majority of commercial distributed applications are based on the client-server model and RPC.

At an abstract level, this project aims to investigate the viability of client server computing for numerical and scientific applications, and the extent to which this paradigm can be reconciled with the message passing model. We are pursuing this via an experimental software system called PVM-RPC, which serves as the vehicle for evaluating both functionality and performance. In this paper, we discuss our motivations and background, and present an overview of the PVM-RPC system as it currently exists; we also provide some performance results comparing PVM-RPC to PVM-MP(message passing) that quantify the (low) overheads introduced. We then describe our plans for future enhancements to the system, including enhanced fault tolerance and load balancing features, and adding an object oriented application programming interface (API) as an extension to the PVM-RPC programming model.

2 Motivations and Goals

RPC systems have been available for a number of years, and most of the issues have been well analyzed and resolved. Nevertheless, enhancements such as combining message passing and RPC, incorporating load balancing with failure resilience, and using request-response paradigms for numerical algorithms can be supported with a system like PVM-RPC. Thus, one goal of this project is to provide a remote procedure call facility to current PVM users, who could potentially benefit from the RPC programming model. Another is to make PVM more suitable for those applications which have traditionally been implemented using the RPC mechanism, and a third is to revisit the RPC model to explore functionality extensions to the RPC paradigm itself. Most existing PVM applications in scientific computing are well suited to the message passing model because of their asynchronous or irregular communication pattern. However, certain software engineering aspects of these applications, such as rapid deployment and robustness, can be enhanced by using the RPC paradigm. Especially since PVM-RPC can co-exist with traditional message passing, portions of an application could be converted to use RPC when appropriate. Simultaneously, traditional client-server applications could benefit from the software infrastructure provided by PVM. We believe that this approach has several advantages over simply using other RPC systems. Several older systems, such as **rpcgen**[8] are cumbersome and time consuming for application development; moreover, they are simplistic in server location mechanisms, provide little fault tolerance, and do not incorporate load balancing schemes. Our goals were to provide these features and more, while minimizing the programmer's effort and system overheads. For our future work we want to carry this philosophy forward while addressing the benefits of newer systems such as Corba [6] which work in an object oriented environment.

3 The PVM-RPC System: Current Status

PVM-RPC[9] is derived from the RPC model[3] in which references to remote services mimic a procedure call. In PVM-RPC, applications design and implement a server which may export one or more services. Clients access these services with invocations that look and behave like procedure calls. However, there are a number of issues which arise since these procedure calls invoke non-local routines; the major issues are name binding, parameter passing, asynchronous processing, and redundancy as it applies to fault tolerance and load balancing. Name binding is the process of mapping the name of a routine as referenced by the client to a service made available by a remote server. While there are many different models for name binding, we use the concept of a service broker (SB) [2, 6], a well-known entity in the distributed computing environment. Servers register with the SB, which publishes available services and from whom clients can obtain the location of a required service provider.

Redundant servers are often desirable to allow for load balancing and failure resilience. PVM-RPC supports multiple servers offering the same service with the RPC invocation mechanisms selecting the best suited server for the initial invocation. In the event of failure during the call, the request is transferred to another server, if possible, without intervention or explicit coding by the programmer. Thus, replicated servers enable simple forms of load balancing and failure resilience in PVM-RPC. For parameter passing, function arguments in PVM-RPC may optionally be tagged by the programmer as being input, output or input-output, depending on whether values are copied from client to server, server to client, or bidirectionally. This feature can help reduce data copying between client and server, particularly in the case of large arrays, while providing the ability to pass large quantities of unidirectional data when needed. PVM-RPC also supports asynchronous remote procedure call; in fact, the API implements a synchronous call by immediately following an asynchronous call with a wait. While an asynchronous invocation diverges from the standard description of RPC as described in [3], in which the invocation of a remote process is described as being “semantically equivalent to a procedure call”. This variation in the RPC model is a common adaptation.

3.1 System Overview

The PVM-RPC system consists of four major components – the pvm daemon (PVMD), the service broker (SB), the servers, and the clients. Of these, PVMD is an integral part of any PVM system, and is responsible for managing the virtual machine, message routing, process initiation, housekeeping, and various other functions. The SB maps service names, as known to clients, to a tuple consisting of a server and service id recognizable by the server. In order to ensure that

PVM-RPC is at least as reliable as PVM with respect to hardware failure, the SB runs on the same machine as the master PVMD. While there are various ways that the SB can start, any server that is trying to register a service will start the SB if it is not already running. When a server starts, it first locates the SB via a table lookup in PVMD. If the SB is not running, then the server starts an SB using *pvm_spawn* and, again, attempts to locate the SB from the PVMD (the second lookup of the SB is done to prevent a race condition between two servers). The server then registers its services with the SB.

Similarly, when a client wishes to use a service, it first locates the SB from the PVMD and then obtains a list of available services from the SB. The SB returns a complete list of all known services and addresses, which the client caches locally in a table; subsequent name resolutions are done by table lookup. When a service becomes unavailable (due either to server failure or congestion) clients update their tables accordingly. If more than one server is available for a particular service, clients interleave requests among all available servers. At each server, requests are processed in FIFO fashion. However, all servers recognize a special "are you alive" message and send an immediate response – thereby permitting clients to determine whether servers are still busy computing their requests or if they have failed. Client stubs, as previously mentioned, always perform asynchronous invocations followed by "wait"s – during which they periodically ping servers and re-issue requests to alternate servers if no "i am alive" responses are received. This simple failure resilience mechanism is of considerable practical value.

3.2 Application Programming Interface

As in any RPC system, PVM-RPC provides constructs for writing servers and clients. To create a server, services must be declared:

```
pvm_rpc_service(service_name[,[IN,OUT,IO] {PVM_TYPE} variable_name, ...])
```

This declaration is then followed by standard C code that defines the implementation of the service. All parameters are treated as pointers. The key words IN, OUT and IO (default) are optionally used to provide hints to avoid data copying where possible. Data types are specified as in PVM, e.g. PVM_DOUBLE or PVM_INT (PVM_CPLX and PVM_DCPLX are currently not supported). For each parameter the programmer declares, there is a 'hidden' integer constant called {*variable_name*}_size (i.e., _size appended to the parameter name) that specifies the number of elements the parameter `variable_name` contains. This variable is generated automatically by the preprocessor. After all services are declared, the statement *pvm_rpc_main* completes the server. A server may itself behave as a client by invoking functions at other servers. While a service may call external functions, all services must be declared in the same file that contains the *pvm_rpc_main* statement.

Clients can invoke a remote procedure either synchronously with the statement *pvm_rpc_sinvoke*, or asynchronously with the statement *pvm_rpc_ainvoke*.

In the asynchronous case, a later *pvm_rpc_wait* statement acts as a synchronization point for the RPC. An example of a client invocation is:

```
pvm_rpc_sinvoke(add_vec,a:SIZE,b:SIZE,c:SIZE);
```

where a, b, and c are declared as `double a[SIZE], b[SIZE], and c[SIZE]`.

3.3 Performance of the PVM-RPC System

In this section we present performance results from a simple benchmark of the current PVM-RPC system. The application computes long range forces among 10000 pairs of particles. The PVM-MP (message passing) program is an adaptation of the "CUBIX CrOS III" algorithm from [5]. In this implementation, the particle space is divided among the available processors (processes in PVM) which are logically connected in a ring. By shuffling each processor's portion of the particle space to ring neighbors, and performing force computations on pairs of particle subspaces during each phase, all-to-all interaction forces are computed. In the PVM-RPC version, servers export services that compute partial forces given two subspaces; a single client repeatedly issues requests to multiple servers with all possible combinations of particle subspaces. Our benchmark exercise was conducted in order to to determine if PVM-RPC incurred significant overheads compared to PVM-MP, while providing a more natural interface to invoke a remote service.

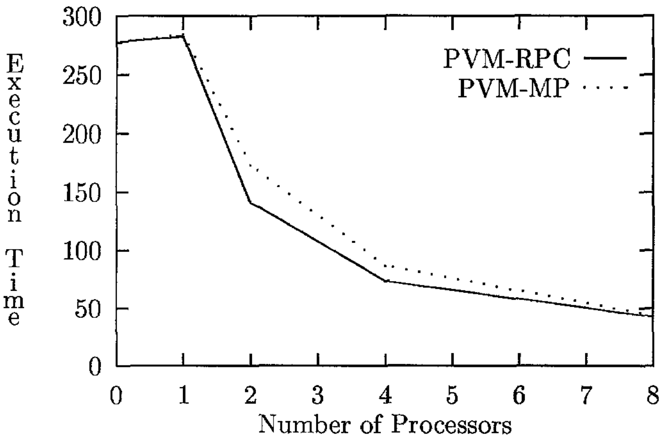

Fig. 1. Scalability of PVM-MP and PVM-RPC.

All programs were compiled with `SunOS cc` version `SC4.0 18 Oct 1995 C 4.0` using the switch -xO4 and run on Sparcstation 20's under SunOS 5.5.1. A series of runs using 0,1,2,4 and 8 processors were performed for both the PVM-RPC and PVM-MP programs, with 0 processors representing the execution time of a sequential program on a single machine. Execution times in seconds for both versions of the program are shown in Fig. 1 for different numbers of processors. Little difference in performance is observed between the two versions; in fact, the RPC version is marginally faster. These results, while only representing an isolated case, nevertheless do indicate that the client-server model does not degrade either programmability or performance for physical science applications exemplified by PIC codes. Moreover, the marginal improvement that RPC exhibits suggests that such a client server organization may be preferable to crowd-computation structures. The reason for the loss in PVM-MP performance is the tight synchrony of the abstract algorithm, which translates poorly to general purpose cluster environments in which processors require slightly different times to complete equal amounts of work. These effects are well-known, and are due to external or operating system perturbations, and occur even when dedicated workstations are used. The RPC paradigm alleviates this synchrony by parceling out work to processors using a bag-of-tasks approach, often (as in this case) resulting in better overall performance. We have also conducted several other benchmarking exercises using textbook programs such as matrix multiply, dot products, and raytracing. In each case, similar results (i.e. performance within 5-10% of PVM-MP) were observed. Although these conclusions cannot be generalized because these applications are not representative of parallel programs with more complex communication patterns, they do indicate to some extent the viability of client-server computing in the scientific application domain.

4 Ongoing Work and Future Plans

As discussed, PVM-RPC incurs little or no performance overhead compared to PVM-MP while enhancing PVM with a client-server framework, some failure resilience and load balancing support, and a procedure call interface for process-to-process communication. However, the current implementation is limited in several respects; we are addressing these and other issues in the belief that these facilities would be valuable complements to the existing system and to native PVM.

4.1 Proposed API

In order to facilitate object oriented applications, a new API is being developed for PVM-RPC and incorporates C++ class definitions as its basis to define remote objects. This syntax is extended by prefixing the keyword *class* with the modifier *distributed* to define an object which requires distributed computation; similar qualification is used to distinguish distributed methods and variables within an object from local counterparts. Although implementation is straightforward, this extension raises tricky issues. For example, if distributed variables are cached on the server, how and when are they updated? Second, as with PVM-RPC, there is an inherent problem with the handling of pointers in that the length of the vector is of an unknown size. Both of these issue could be addressed by using a newly created PVM class type which require all modifications to distributed variables to both dirty the cache and adjust a hidden size modifier. However, this approach would introduce significant overhead and requires further study.

Another API extension concerns non-blocking invocations. We are considering (possibly nested) COBEGIN-COEND structures that enable client side threads to be dynamically created to simulate efficient fork-join operations. Using this feature, a distributed calls could take place simultaneously. Such a feature will also enable the implementation of asynchronous callbacks or notification mechanisms. An example of a possible notation is:

```
ptr.add(a:20,b:40:2,c): NOTIFY add_handler(c)
```

This example illustrates a remote method add which will return prior to completion of the remote computation. When the remote computation has completed the routine `add_handler` will be called with argument `c`. Since distributed classes will be converted to C++ classes by a preprocessor, normal C++ features such as inheritance and virtual functions are supported.

4.2 Fault Tolerance and Load Balancing

In PVM and other metacomputing environments, the ability to continue a job despite individual machine failures is very beneficial. PVM-RPC, like Sciddle[1] can only detect server failures at synchronization points (i.e. *pvm_rpc_sinvoke* or *pvm_rpc_wait*) and re-issues requests. Unlike Sciddle, PVM-RPC attempts to handle load balancing and fault tolerance automatically. Therefore, increased benefits can be accrued if clients are able to detect failure immediately. To facilitate this we are attempting to incorporate this improvement by using a multithreaded client calling mechanism.

While the current PVM-RPC strategy of multiplexing requests among multiple servers works well if machines are homogeneous and there is only one user, it is not as effective in heterogeneous systems with disparate server capabilities or in an open environment with multiple clients share the same servers. To improve load balancing in these situations a more sophisticated system is needed which allows the client to obtain some information regarding the current load on a server. We intend to address this problem by

- enhancing servers to report their current load when responding to a 'ping';
- allowing the service broker to cache current load from 'ping' reports;
- permitting the client to query the service broker for unused servers while waiting for a long request to complete.

Effective load balancing is a difficult task because the load at a server is dynamic and a particular technique which is optimal for one class of users may be non-optimal for another. Excessive polling of the server to obtain load information can artificially inflate the load while delayed polling of the servers can result in execution delays. Furthermore, a policy found effective for one type of computation might be ineffective for another since request size, execution time, and network bandwidth all affect the viability of shifting a request from one server to another. Based on these issues, we are incorporating a load-balancing strategy in which clients make decisions based on instantaneous conditions and on queries of the SB and servers.

5 Conclusion

The current version of PVM-RPC has given us some insight into the benefits and drawbacks of the system design, functionality, and performance. Although efficiency and performance are satisfactory, the API, tolerance to failures, and the load balancing mechanism can be improved. We are addressing these issues and have begun to implement our proposed solutions. We are also conducting more extensive tests on a number of other classes of applications, to ensure that the RPC paradigm is indeed suitable in a variety of circumstances. We expect that successful completion of these investigations will make PVM-RPC a viable and valuable platform for parallel distributed computation.

References

1. Peter Arbenz, Martin Billeter, Peter Guntert, Peter Luginbuhl, Michela Taufer, and Urs von Matt. Molecular dynamics simulations on cray clusters using the sciddle-pvm environment. *Parallel Virtual machine - EuroPVM'96*, pages 142–149, October 1996.
2. A. D. Birrell. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 4(15):260–273, 1982.
3. A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems*, 2:39–59, 1984.
4. H. Casanova and J. Dongarra. Netsolve: A network solver for solving computational science problems. Technical Report CS-95-313, University of Tennessee, November 1996.
5. G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors.* Prentice Hall, Englewood Cliffs, 1988.
6. Object Management Group. *The Common Object Request Broker: Architecture and Specifications, 2.0 (draft) ed*, May 1995.
7. V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.
8. *SunOS Reference Manual.* Mountain View, California, 1990.
9. A. Zadroga, A. Krantz, S. Chodrow, and V. Sunderam. An rpc facility for pvm. *High-Performance Computing and Networking '96, Brussels, Belgium*, pages 798–805, April 1996.