Client Server Computing on Message Passing Systems: Experiences with the RPC Model in PVM ⋆

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, based on an experimental RPC framework for the PVM system. The project investigates the potential for RPC to complement asynchronous message passing, thereby making the request-response paradigm available 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. The essence of this subsystem, and its associated programming interface, is a facility for PVM that permits the specification and export of *services* that may be invoked by clients using the well-established RPC paradigm. 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.

Keywords: PVM, Client-Server, Object Oriented, RPC

1 Introduction

Parallel Virtual Machine (PVM)[7] is a widely-used software system for heterogeneous parallel and distributed computing. PVM supports a concurrent computing model based on multiple processes cooperating via the exchange of strongly typed messages. Its implementation via daemon processes and user libraries, permits the execution of such parallel programs in multifaceted hardware environments. Such environments tend to be distributed computing platforms comprised of servers, workstations, and other computers connected by general purpose networks, although PVM will also work on hardware multiprocessors. With the increasing technological and economic viability of desktop workstations, distributed computing with PVM and similar systems has become commonplace, both for traditional applications as well as for high performance parallel applications.

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. Nevertheless, it has become the paradigm of choice for parallel applications, especially in scientific domains, and is embodied in a number of systems and libraries, such as PVM, MPI, NX, and CMMD.

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 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, 4] 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 investigating 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

⋆ 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. ASC-9214149 and ASC-9527186.

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 PVM-RPC 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) for PVM-RPC as a means of further abstracting the client-server relationship.

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, a few useful features and situations, such as combining message passing and RPC, incorporating failure resilience, and using request-response paradigms for numerical algorithms could result from a system like PVM-RPC. There are three compelling reasons for our design and implementation of this system. The first is to provide an RPC facility, with its well understood abstractions and resulting benefits, to current PVM users, who could potentially benefit from a more convenient programming model without loss of performance. The second is to make PVM more suitable for those applications which have traditionally been implemented using the RPC mechanism. The third is to revisit the RPC model and look for methods to provide an RPC mechanism which is more transparent to the user, both when developing and executing their applications, than current systems.

There are currently many users of PVM in the scientific community who make effective use of PVM message passing facility. While many typical PVM applications in scientific computing are well suited to the message passing model because of their asynchronous or irregular communication pattern, we feel that software development could be faster and more robust in a number of cases, 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. Further, 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 no load balancing. Our goals were to provide these features and more, while minimizing programmer effort and system overheads. For our future work we want to carry this philosophy forward while addressing the benefits of newer systems such as [6] which work in an object oriented environment.

3 The PVM-RPC System: Current Status

PVM-RPC[9] is derived from the RPC model[4] in which references to remote services mimic a procedure call. Applications design and implement a server which may export one or more services. These services can then be accessed by a client via an invocation that looks and behaves like a procedure call. However, there are a number of issues which arise since the procedure call invokes a non-local routine, including name binding, parameter passing, asynchronous processing, and redundancy as it applies to fault tolerance and load balancing. Our model need not consider issues related to the transport layer since PVM already addresses these concerns.

Name binding is the process of mapping the name of a routine 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) [3, 6]. The SB is a process that maps a service, as named by a client, to the server that provides the service. Redundant servers are often desirable to allow for load balancing and failure resilience. In our model, we allow redundant servers to exist, and if they do exist, the mechanism that invokes a remote procedure will detect failure and transfer the request to another server if possible. If no alternative server exists, the client is informed that the service is unavailable. Replicated servers also enable failure resilience – client calls can be tagged to return error indications when a server fails; applications need only repeat that particular call which will be automatically routed to a different server.

For parameter passing, PVM-RPC allows variables to be tagged as input, output or input-output, depending on whether values are copied from client to server, server to client or bidirectionally. This feature is optional and is implemented to reduce unnecessary copying of values between client and server, particularly in the case of large arrays, while at the same time providing the ability to pass large quantities of data in one direction when needed.

PVM-RPC provides two types of remote procedure calls, synchronous and asynchronous. While an asynchronous invocation diverges from the standard description of RPC as in [4], in which the invocation of a remote process is described as being "semantically equivalent to a procedure call", this variation in the model is extremely valuable, especially since it enables concurrency at the client end. In PVM-RPC, parameter copying by the system at invocation and synchronization (wait) points helps reduce added programmer housekeeping that is otherwise required.

3.1 System Overview

PVM-RPC consists of four major components – the pvm daemon (PVMD), the service broker (SB), the servers, and the clients. The remainder of this section describes the function and interaction of these components. PVMD is an integrated part of any PVM system, and is responsible for identifying and managing global resources, such as available hosts, message routing, and authentication.

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. A complete scenario of the SB interaction with the rest of the PVM-PRC system is now illustrated: When a server starts, it first locates the SB from the PVMD using the table facility. 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 finally registers its services with the SB and no further communication between the server and the SB is required.

Likewise, when a client wishes to use a service, it first locates the SB from the PVMD and then gets a list of available services from the SB. In the client's case, a request for a list of registered services results in a complete list of all known services and addresses, which the client caches locally to eliminate unnecessary communication between the client and SB. While the list of servers and services may change during the client's execution, this is not an issue unless the client requires a service whose address is not cached, or there is a problem with a known provider of a service (due either to server failure or congestion). In either of these cases, the client can update its list of known services. If more than one server is available for a particular service, the client by default will interleave requests among all available servers.

Servers are entities which provide access to remote resources and a method to parallelize computations. In PVM-RPC, a server registers its services with the SB and then listens for client's requests. Generally, requests are queued and processed in a 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. While the PVM-RPC interface provides both a synchronous and asynchronous call, both are internally asynchronous. That is, an RPC returns as soon as a message has been sent by the client to the server, with a tag identifying the RPC. To ensure that the call is completed, at a later time, a wait is done on the tag. To provide fault tolerance during this wait operation, a message is sent periodically to the server to determine the server's status. If there is no response from the server to this message, the wait operation assumes the server has died and sends a new request to another server. A synchronous RPC is implemented as an asynchronous RPC followed by an immediate wait.

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 and coded, using a syntax similar to standard C. The primitive *pvm_rpc_service* is used to declare a service. Its syntax is:

```
pvm_rpc_service(service_name,[[IN,OUT,INOUT] {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, with array notation favored for element access. The key words IN, OUT and INOUT are used to give the server a hint as to whether variables need to be copied from client to server (IN), server to client (OUT) or bidirectionally (INOUT). If none of IN, OUT or INOUT is specified the variable is assumed to be INOUT. The type of a parameter is PVM_TYPE, which must be specified, and must be a legal PVM type such as PVM_INT (PVM_CPLX and PVM_DCPLX are currently not supported). Finally, for each parameter the programmer declares, there is a 'hidden' integer constant with the name {*variable_name*}_size (i.e., _size appended to the parameter name) that specifies the number of elements the parameter named `variable_name` contains. This variable is automatically generated by the preprocessor to reduce the number of parameters the service would otherwise require. A complete example demonstrating the usage of *pvm_rpc_service* is shown in Fig. 1.

```
pvm_rpc_service(add_vec,IN PVM_DOUBLE a,IN PVM_DOUBLE b,OUT PVM_DOUBLE c) {
    int i;
    for(i=0; i < a_size; i += 1) {
        c[i] = a[i] + b[i];
    }
}
```

Fig. 1. Example showing how to declare and implement a service that sums two vectors.

After all services are declared, the server is completed with a call to *pvm_rpc_main*. This must be the last statement in the file. This statement will cause the preprocessor to generate a main program containing the code for registering the defined services and processing requests from clients. There are no restrictions on the number of services a server may contain, nor on references to external functions by a service other than those imposed by C. A server may in itself behave as a client by invoking functions from 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. Figure 2 shows a complete server containing one service, *add_vec*.

```
#include <pvm3.h>
#include <pvm_rpc.h>
pvm_rpc_service(add_vec,IN PVM_DOUBLE a,IN PVM_DOUBLE b,OUT PVM_DOUBLE c) {
    int i;
    for(i=0; i < a_size; i += 1) {
        c[i] = a[i] + b[i];
    }
}
pvm_rpc_main;
```

Fig. 2. Example of a complete PVM-RPC server.

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. To limit unpredictable behavior, parameters used in an RPC are not modified by the PVM-RPC system until completion of the remote procedure (i.e, the parameters are only modified during the *pvm_rpc_wait* statement). An example of a client invoking an RPC 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 metrics of the PVM-RPC System

In this section we present a performance metric of the current PVM-RPC system by benchmarking a program (see appendix A) which multiplies two matrices by utilizing a server for multiplying vectors. Each vector is multiplied 100000 times to simulate coarse grain parallelism. A similar PVM-MP program, shown in appendix B, which uses a slave process to perform the vector multiplications, was run for comparison with the PVM-RPC program. The purpose of this benchmark is to demonstrate that the PVM-RPC layer does not incur a significant overhead compared to the PVM-MP system, while providing a more natural interface to invoke a remote service. Other similar programs (not shown here) indicate that for finer grain programs PVM-RPC can sometimes reduce the elapsed execution time by small amounts, when compared to a PVM-MP program, through the automated multiplexing of network communication with computational tasks. On average the PVM-RPC program did require 0.10 seconds of additional CPU time when compared to the equivalent PVM-MP program. This overhead is probably due to parameter marshaling and fault tolerance code.

All programs were compiled with `gcc` version 2.6.3 using the switch -O2 and run on a combination of Sparc IPC's and 1+'s under SunOS 5.5.1. To show scalability, a series of runs using 0 to 6 processors were performed on both the PVM-RPC and PVM-MP programs with 0 processors representing the execution time of a sequential program on a single machine and 1 to 6 processors representing the number of slaves or clients. The relative performance of these two programs for different vector lengths are shown in Fig. 3 with the Y axis representing the wall clock time in seconds, and the X axis representing the number of processors. Timing of the programs began on the first call to the multiplication procedure and terminated when the last result was returned. That is, initialization code was not included in the timings.

As expected, both programs show near linear speed up when a second processor is added and diminishing returns as additional processors are added. Furthermore, both the PVM-MP and PVM-RPC programs had near identical times, though what is surprising is that the PVM-RPC program outperforms the PVM-MP program on finer grain programs (not shown here). This can be explained by noting that the PVM-RPC server is multi-threaded and therefore incoming requests can be queued up as the program is executing. That is transmitting time of the next request can be overlapped with the processing of the current request. Although PVM itself is not thread safe, PVM-RPC server dispatch and other generic functions have been implemented in such a manner that servers are multithreaded. In the graph, the plateau seen for 3, 4, and 5 processors occurs because at least one slave or client must multiply two vectors in all three cases making the wall-clock time nearly identical for these three cases. Similar close correlations were observed in other exercises, indicating that PVM-RPC adds little or no overhead to PVM-MP and scales as well.

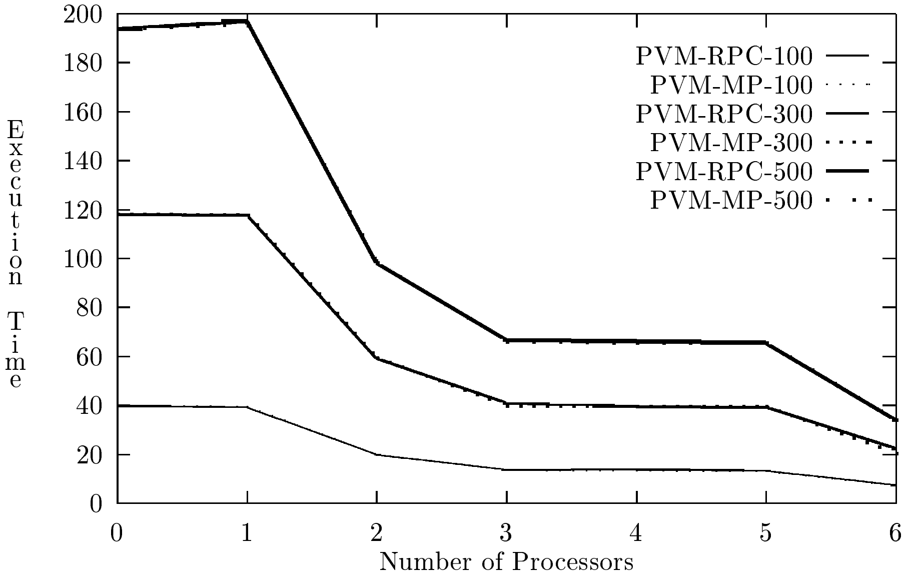

Fig. 3. Scalability of PVM-MP and PVM-RPC.

4 Ongoing Work and Future Plans

Our current implementation of PVM-RPC incurs little or no performance overhead compared to PVM-MP while adding, to PVM, a client-server facility, fault tolerance, and a procedure call interface for process-to-process communication. However, there are still a number of limitations we are trying to resolve:

- The current API of PVM-RPC is unsuitable for an object oriented paradigm or event driven programming;
- Fault tolerance is somewhat limited and static in nature;
- PVM-RPC currently does not support dynamic load balancing.

The addition of these features should create an environment suitable for the implementation of distributed systems, such as Netsolve[5]. In this section we describe our plans to extend PVM-RPC's API to an object oriented paradigm; improve dynamic fault tolerance and add an extended load balancing facility which will use the underlying system as described in section 3.1.

4.1 Proposed API

The new API being developed for PVM-RPC 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. Since this class can contain both local and distributed methods, those methods which require distributed computation and are to be implemented in a server are prefixed with the modifier *distributed*. Likewise, those variables local to the class definition which are required by the server are also prefixed with the modifier *distributed*. For example to define a class to perform distributed operations on vectors the following definition could be used:

```
distributed Class math {
    public:
       distributed void add(in PVM_DOUBLE a, in PVM_DOUBLE b, out PVM_DOUBLE c);
       distributed void sub(in PVM_DOUBLE a, in PVM_DOUBLE b, out PVM_DOUBLE c);
}
```

In this example, a class containing two methods for performing non-local operations add and sub have been declared. They have as input two vectors, `a` and `b`, and as output a single vector `c`. However, if it is desired that all vector operations be performed on two fixed vectors, the following definition would be used:

```
distributed Class math {
       distributed double v1[100],v2[100];
    public:
       math(v1,v2);
       distributed void add();
       distributed void sub();
}
```

While, implementation of this modification to the class structure is straight forward, a number of complicated issues are raised: First, if the distributed variables are cached on the server, to prevent multiple sending of data during invocation, how do you indicate in an efficient manner when the variable needs to be resent? 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. We will, however, allow a stride and size modifier to be included as part of the function call to enable less cumbersome usage of dynamic sized vectors. The complete syntax for this will be `var[:size[:stride]]` with the default values for size and stride being 1. So, for example, using the first definition of `math` we could invoke the function `add` with:

```
math ptr;
ptr.add(a:20,b:40:2,c);
```

To allow asynchronous calls of remote functions which do not block the caller, a syntax is added to enable threads to be dynamically created to simulate an efficient fork-join operation. While the exact syntax for this mechanism is still being adjusted, one possibility is:

```
COBEGIN
    ptr.add(a:20,b:40:2,c);
```

```
AND
    ptr.add(d:20,e:20,f);
COEND
    ptr.add(f:20,c:20,f);
```

In this example, a distributed call could take place in one code branch while the other code branch executes another distributed call. Naturally, `COBEGIN-COEND` sequences can be nested. Since an active thread will be used to process the RPC it will be possible to implement callbacks or notification of completion of an asynchronous call. The syntax for this still needs to be worked out but one possibility is:

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

In this example the add routine returns immediately and the routine `add_handler` is called with argument `c` when add has completed. Since distributed classes will be converted to C++ classes by a preprocessor, normal C++ features such as inheritance and virtual functions are supported. This allows for the user to create objects consisting of one or more distributed objects or to override a distributed function with a local one. These modifications to the current API will not only allow the user to create distributed objects (using C++) but will also enable the implementation of other features such as event driven programming and enchanced fault tolerance.

4.2 Fault Tolerance

When utilizing many workstations, transitional failures are not uncommon. The ability to continue processing during a transitional failure can result in a significant savings of wall-clock time. PVM-RPC, like Sciddle[1, 2] defines explicit synchronization points for asynchronous calls. While there are advantages in this design, a disadvantage is that fault tolerance detection only occurs when the client is waiting for a reply at a synchronization point, *pvm_rpc_sinvoke* or *pvm_rpc_wait* in PVM-RPC. Even though there is minimal loss of wall-clock time between the detection of a failed RPC and a new request being submitted during a *pvm_rpc_sinvoke*, there can be a significant loss in time if failure detection of a failed *pvm_rpc_ainvoke* does not occur prior to the *pvm_rpc_wait*. Another drawback in the current implementation of PVM-RPC fault tolerance is that the client 'pings' the server at fixed intervals. If the server fails to respond in a finite amount of time, the server is considered failed and the request is sent to another server. While this technique works, it has a few draw backs which are being addressed; some techniques we are implementing to improve fault tolerance are mentioned below.

First, time outs should be dynamic and based upon the required (or estimated) processing time of the request. Initially, we will try a step off scheme where the client doubles the wait time between pings. Second, the client should be able to detect failure at the time of failure during an asynchronous call instead of waiting for a synchronization point. This shortcoming is being addressed by changing the implementation of the client's calling mechanism to use threads during asynchronous invocations. This modification will allow the client to detect failure of the server before the synchronization point and hence reduce the wait time because the request can be forwarded to another server earlier. Third, the current system has a critical point at the service broker. We are looking into ways to reduce dependency on this critical component.

4.3 Load Balancing

While the current PVM-RPC strategy of multiplexing requests among multiple servers works well if machines are homogeneous and there is only one user, this strategy is not as effective in either a heterogeneous system where systems have different processing power and some systems have multiple processors, or in an open environment with multiple clients sharing 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 will address this problem by

- allowing servers to report their current load when responding to a 'ping';
- allowing the service broker to cache current load from 'ping' reports;
- allowing 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 excessive execution delay since a request is allowed to proceed on a slow server before being restarted on a faster server. 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.

In our basic design, the decision to move a request from one server to another will be made by the client. Furthermore, the client can obtain the instant load of a server by pinging it and an estimate of a server load by querying the service broker. Using this design we will explore several techniques including modifications to query time and decision ranging from canceling a request in progress to using load information as a determining factor of where to send future requests.

5 Conclusion

The current version of PVM-RPC has given us some insights into the benefits and drawbacks of the PVM-RPC protocol. While we are pleased with the computational performance of the current system, the current API and limited fault tolerance can be improved. Furthermore, the current implementation of PVM-RPC does not employ any facility for load balancing. We have begun to address these issues and are beginning to implement our proposed solutions. The new API being developed not only provides an object oriented syntax but also address issues related to vector manipulation, event driven programming and provides a base for implementing an enhanced fault tolerance system and load balancing.

A PVM-RPC program to multiply two matrices

```
#include <pvm3.h>
#include "pvm_rpc.h"
#include <signal.h>

pvm_rpc_service(vdot,IN PVM_DOUBLE a,IN PVM_DOUBLE b,OUT PVM_DOUBLE c) {
    int i,j;
    for(j=0; j < 100000; j++) {
        c[0] = 0;
        for(i=0; i < a_size; i += 1) {
            c[0] += a[i] * b[i];
        }
    }
}

pvm_rpc_main
```

```
#include <pvm3.h>
#include "pvm_rpc.h"
#include "../my_time.h"
#define A_ROW 2
#define A_COL 100
#define B_ROW A_COL
#define B_COL 3

double a[A_ROW][A_COL] = {
    {3,1,-1,...},
    {2,0, 3,...},
};
double b[B_COL][B_ROW] = {
    {1,3,2,...},
    {4,-1,5,...},
    {6,9,1,...},
};

double c[A_ROW][B_COL];
PVM_RPC_TAG tags[A_ROW][B_COL];

main() {
    int i,j;
    pvm_setopt( PvmRoute, PvmRouteDirect );
    start_time;
/*
 * Start A_ROW * B_COL vector dot products
 */
    for(i=0; i < A_ROW; i++) {
        for(j=0; j < B_COL; j++) {
            pvm_rpc_ainvoke(vdot,tags[i][j], a[i],A_COL,b[j],B_ROW,&c[i][j],1);
        }
    }
```

```
/*
 * Wait for the completion of A_ROW * B_COL vector dot products
 */
    for(i=0; i < A_ROW; i++) {
        for(j=0; j < B_COL; j++) {
            pvm_rpc_wait(tags[i][j]);
        }
    }
    print_time(1);
    for(i=0; i < A_ROW; i++) {
        for(j=0; j < B_COL; j++) {
            printf("%10.5lf ",c[i][j]);
        }
        printf("\n");
    }
}
```

B PVM-MP program to multiply two matrices

```
#include <stdio.h>
#include "pvm3.h"

main() {
    int ptid,oldval,len,i,j,msgtag,bufid,mtid;
    double sum,*v1,*v2;
    oldval = pvm_setopt( PvmRoute, PvmRouteDirect );
    ptid = pvm_parent();
    mtid = pvm_mytid();
    pvm_initsend(PvmDataDefault);
    while(1) {
        bufid = pvm_recv(-1,-1);
        pvm_bufinfo(bufid, NULL, &msgtag, &ptid);
        pvm_upkint(&len,1,1);
        if(len == 0) {
            pvm_exit();
            exit(0);
        }
        v1 = (double *)malloc(sizeof(double)*len);
        v2 = (double *)malloc(sizeof(double)*len);
        pvm_upkdouble(v1,len,1);
        pvm_upkdouble(v2,len,1);
        for (j =0 ;j <100000 ;j ++) {
            sum = 0;
            for(i=0; i < len; i++) {
                sum += v1[i] * v2[i];
            }
        }
        pvm_initsend(PvmDataDefault);
        pvm_pkdouble(&sum,1,1);
        pvm_send(ptid,msgtag);
        free(v1);
        free(v2);
    }
```

```
    pvm_exit();
    exit(0);
}

#include <stdio.h>
#include "pvm3.h"
#include "../../my_time.h"

#define A_ROW 2
#define A_COL 100
#define B_ROW A_COL
#define B_COL 3

double a[A_ROW][A_COL] = {
    {3,1,-1,...},
    {2,0, 3,...},
};
double b[B_COL][B_ROW] = {
    {1,3,2,...},
    {4,-1,5,...},
    {6,9,1,...},
};

double c[A_ROW*B_COL];
#define MAX_PROC 10

main(argc,argv)
char **argv;
int argc;
{
    int oldval,tids[MAX_PROC],cc,i,j,bufid,size,msgtag,sender;
    int N_PROC;

    oldval = pvm_setopt( PvmRoute, PvmRouteDirect );
    N_PROC = atoi(argv[1]);
    fprintf(stderr,"Using %d slaves\n",N_PROC);

    cc = pvm_spawn("/home/atk/work/pvm/pvm3/bin/SUN4SOL2/mult_slave",
                   (char**)0, 0, "", N_PROC, tids);
    if (cc != N_PROC) {
        fprintf(stderr,"Error code %d in starting slave process.\r\n",cc);
        exit(0);
    }
    start_time;
    for(i=0; i < A_ROW; i++) {
        for(j=0; j < B_COL; j++) {
            pvm_initsend(PvmDataDefault);
            size = A_COL;
            pvm_pkint(&size,1,1);
            pvm_pkdouble(a[i],A_COL,1);
            pvm_pkdouble(b[j],B_ROW,1);
            pvm_send(tids[(i*B_COL+j)%N_PROC],i*B_COL+j);
        }
    }
```

```
    for(i=0; i < A_ROW; i++) {
        for(j=0; j < B_COL; j++) {
            bufid=pvm_recv( -1, -1 );
            pvm_bufinfo(bufid,&size,&msgtag,&sender);
            pvm_upkdouble(&c[msgtag],1,1);
        }
    }
    print_time(1);
    for(i=0; i < A_ROW; i++) {
        for(j=0; j < B_COL; j++) {
            printf("%10.5lf ",c[i*B_COL+j]);
        }
        printf("\n");
    }
    pvm_exit();
}
```

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. Peter Arbenz, Christoph Sprenger, Hans Peter Luthi, and Stefan Vogel. Sciddle: A tool for large scale distributed computing. Technical Report 213, Eidgenossische Technische Hochschule Zurich, Department Informatik Institut fur Wissenschaftliches Rechnen, March 1994. `ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/2xx/213.ps.gz`.
3. A. D. Birrell. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 4(15):260–273, 1982.
4. A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transaction on Computer Systems*, 2:39–59, 1984.
5. H. Casanova and J. Dongarra. Netsolve: A network solver for solving computational science problems. *Super–Computing 96*, 1996.
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.