A PERFORMANCE EVALUATION OF RPC, JAVA RMI, MPI AND PVM

Kalim Qureshi
Department of Math and Computer Science
University of Kuwait
Kuwait
email: qureshi@sci.kuniv.edu.kw

Haroon Rashid
COMSATS Institute of Information Technology
Abbattabad
Pakistan
email: haroon@ciit.edu.pk

ABSTRACT

This paper presents performance comparison of Remote Procedure Calls (RPC), Java Remote Machine Invocation (RMI), Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). The Bandwidth, latency and parallel processing time are measured using standard benchmarks. The results show that the MPI performance is much closer to RPC performance.

Keywords: ***Performance Evaluation; Message-Passing Paradigm, MPI, PVM, RPC, Java RMI***

1.0 INTRODUCTION

In a distributed memory system, where memory and address spaces are local to each processor, data can only be shared among processes by message passing. Therefore, the efficiency of message passing library is particularly important in achieving performance improvement. There exit many message passing libraries today. The popular ones include the Message Passing Interface (MPI) [1-4], Parallel Virtual Machine (PVM) [5], Express [6], p4 [7] and Java RMI [8]. These environments enable a collection of heterogeneous computers to be used as a coherent and flexible parallel computing resource. Thus, large computational problems can be solved by using the aggregated power of many computers. In this paper, we address the features of RPC, PVM, MPI and Java RMI. RPC is used as the reference paradigm for performance comparison.

This paper is organized as follows. In section 2, we present the architecture of RPC, PVM, MPI and JAVA RMI, highlighting the features for comparison. Section 3 provides the implementation details and the experimental setup. The results of the comparative study are discussed in section 4 followed by some conclusions drawn in section 5.

2.0 ARCHITECTURE

The purpose of this section is to present the architecture of RPC, PVM, MPI and Java RMI so that this paper is self-contained and to provide additional insights into the features of message passing environments.

2.1 Remote Procedure Call (RPC)

The remote procedure calling (RPC) is a back-bone of almost all distributed software packages and is an implicit part of all flavors of UNIX operating systems. It has the ability to distribute parts of a program to other computers on a network. An RPC facility manages the exchange of data between computers to make remote execution transparent to the user. It supports heterogeneity of the hardware and operating system. RPC supports two kinds of libraries to build distributed application i.e. low-level library and other is high-level library support [9].

By using RPC, programmers of distributed applications avoid the details of the interface with the network. RPC uses the client/server model. The requesting program is a client and the service-providing program is the server. Like a regular or local procedure call, an RPC is a synchronous operation requiring the requesting program to be suspended until the results of the remote procedure are returned. However, the use of threads that share the same address space allows multiple RPCs to be performed concurrently. The RPCGEN is a tool, which generates remote program interface modules. It compiles the source code written in the RPC Language. RPC Language is similar in syntax and structure to C. RPCGEN produces one or more C language source modules, which are then compiled by a C compiler [9].

2.2 Parallel Virtual Machine (PVM)

The PVM system consists of the daemon (or pvmd), the console process and the interface library routines. One daemon process resides on each constituent machine of the virtual machine. Daemons are started when the user starts PVM by specifying a host file, or by adding hosts using the PVM console. The PVM console is the interface for users to interact with the PVM environment. The console can be started on any machine of the virtual machine. Using the console, a user can monitor the status of the PVM environment or reconfigure it. The final component, the PVM interface library, has routine for message-passing, spawning of application processes and coordination of these processes.

The daemons in PVM play a pivotal role. Besides being responsible for spawning new processes and fault-detection, a daemon is also the message router. All communication between applications processes will normally be brokered by the daemons since by default, each application process can only communicate directly with the local daemon. Each daemon maintains information concerning the location and status of all application processes in the virtual machine. When a message for a route process is received from a local application process, the local daemon determines the physical location of that remote process and forwards the message to the remote daemon.

Both UDP and TCP sockets are used in PVM. UDP sockets are set up between a pair of daemons and between a daemon and a local task. The daemon-daemon socket is used for carrying data and inter-daemon control. When a user starts a daemon, the daemon sets up a single TCP socket with each daemon in virtual machine. These TCP carry standard-out and standard-error messages back to the user.

PVM also provides the set-opt function [10] to enable application to communicate directly with each other via TCP sockets. Improvements of two to three times in the transmission time can be achieved, beside better reliability of data transmission. However, there is a large overhead associated with the setting up of TCP sockets. Furthermore, as the number of sockets that can be created is limited on each machine, PVM system will revert to daemon-task communication at some stage.

PVM requires a three-step procedure for a task to send (or receive) a message. For sending a message, a send buffer has first to be initiated, then the data is packed into the buffer, and finally the data in the buffer is sent. Later versions e.g. (PVM 3.3) provide the option to send data using a single call.

2.3 Message Passing Interface (MPI)

MPI is intended to be a standard message-passing specification that each Massively Parallel Processors (MPP) vendor would implement on his or her system. The MPP vendors need to be able to deliver high performance and this became the focus of the MPI design [11]. Reported performance comparison studies [12, 13] show that PVM and MPI have highly comparable performance on the Cray T3D and IBM SP-2. MPI contains the following main features:

- A large set of point-to-point communication routines (by far the richest set of any library to date),
- A large set of collective communication routines for communication among groups of processes,
- A communication context that provides support for the design of safe parallel software libraries,
- The ability to specify communication topologies,
- The ability to create derived data types that describe messages of non-contiguous data.

The PVM and MPI are based on message passing paradigm therefore, the short summary of the comparison [14] is illustrated in Table 1.

Table 1: Similarities between manipulating communicators in PVM and MPI

Function	PVM	MPI
Communicator creator	pvm_staticgroup	MPI_COMM_CREAT MPI_INTERCOMM_CREAT
Communicator destruction	Pvm_lvgroup	MPI_COMM_FREE
New context	Pvm_newcontext	MPI_COMM_DUP
Inter-communication	No restrictions	Some restrictions
Support fault tolerance	yes	no

MPI provides an interface for the basic user, yet is powerful enough to allow programmers to use the high performance message passing operations available on advanced machine. When a MPI-program is executed on a master process, the function *MPI_Init()* initializes the execution environment. It opens a local socket and binds it to a port and verifies that communication can be established with other processes. Then, it distributes MPI internal state to each task so that they start execution. Each task will execute its own portion of code and when it finishes, it will call *MPI_Finalize()* which closes the communication library and terminates the service. Both *MPI_Init()* and *MPI_Finalize()* must be executed by each task. MPI has several kinds of send and receive functions to facilitate many applications requirements.

2.4 Java Remote Method Invocation (Java RMI)

The remote method invocation (RMI) framework [15-18] provides an intermediate network layer that allows Java objects residing in different virtual machines to communicate using normal method calls. This means that a client should be able to access a server on the local machine or on the network as if they were executing in the same run-time system (see Fig. 1) and the networking details necessary for the distributed application were not required. To have a remote object accessible, an interface must be defined which declares those methods that we wish to make public. The server must implement this interface, and any other interfaces it needs. A stub and skeleton must then be generated using RMI, a tool available in JDK. The stub is a class that automatically translates remote method calls into network communication setup and parameter passing. The skeleton is a corresponding class that resides on the remote virtual machine, and which accepts these network connections and translates them into actual method calls on the actual object. This remote object must be registered with a naming service that allows clients to locate where it is running. The client connects to a naming registry and asks for a reference to a service registered under the name registered in the server. The naming registry then returns a remote reference to the object that is used in the client as a normal object.

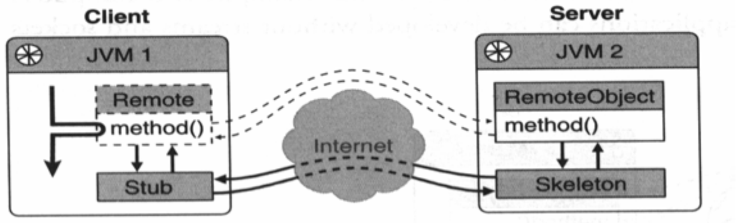

Fig. 1: RMI architecture

3 IMPLEMENTATION

For this benchmarking, eight Intel PIII, 333Mhz (loaded with Linux Operating System) workstations connected by 10 Mbps Ethernet network were used. The Java Development kit used is version 1.2, PVM 3.3, MPI 2 and RPC ONC SUN is the default part of Linux/Free-BSD and many other similar operating systems. To measure the bandwidth and latency, we developed matrix multiplication benchmark program in MPI, PVM, RPC and Java RMI. The master machine initializes two arrays with floating point numbers. Then, it sends the two arrays one after the other to the worker/node. The worker will receive the two arrays accordingly and multiplies them. The result of the multiplication is sent back to the master. When the master receives the result, it will print it and compute the time that was taken to send and receive the matrices. The measured time is used to compute the bandwidth using the equation given below:

$$\frac{64 * (sizeof\ (matrix1) + sizeof\ (matrix2) + sizeof\ (resultmatr\ ix))}{time\ elapsed}$$

The latency and bandwidth of RPC, MPI, PVM and Java RMI paradigms are measured by varying the packet size (in bytes) as follows: 0, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, and 10000.

Table 2: Summarizes the comparison of RPC, PVM, MPI and Java RMI.

Features	MPI	PVM	RPC	Java RMI
Portability	Portable on famous massively parallel computing system and cluster of PCs/workstations	Portable on few massively parallel computing system and cluster of PCs/workstations	Built-in part of all UNIX OS	cluster of PCs/workstations Machine need JVM to run RMI applications
Process creation	Static and dynamic (latest version)	Static and dynamic	Static and dynamic	Static
Control & management of messages sent	Does not support	Does not support	Does not support	Supports e.g. exceptions, security manager
Topology support (Ring, Mesh, Hypercube e.t.c.)	Supported	No	No	No
Load balancing Lib. support	Excellent	Good	No	No

To measure the performance of each paradigm under a master and number of workers configuration, we used Embarrassingly Parallel (EP) benchmark [19] that is the matrix multiplication of 1024x1024. The one row and one column were used as a unit of task and tasks were distributed at runtime (as the assigned task is completed by the node/worker, master will assign a new task) with fixed subtask size. The total number of tasks processed by each node is dependent on the node's performance. The performance was measured and run on a cluster of PCs with the following hardware and software:

Table 3: Distributed system specification

Hardware	8 PIII nodes
Processor Speed	333 MHz
Memory	1.0 GB / node
Interconnect	10 Mb/s
Operating System	Linux 2.4.9-31
Parallel Environment	RPC, MPI, PVM & Java RMI

We have taken care to ensure that no user processes other than the program being benchmarked are using the resources. The programs are also run a number of times and the average results are recorded.

4 RESULTS AND DISCUSSION

In this comparative study, the RPC was used as a reference for comparison because most of the distributed programming packages including the message passing paradigms are based on RPC. Since RPC is the part of UNIX operating system so for its execution no special installation or execution of daemon is needed. Therefore, we have performance priority in terms of latency, bandwidth and total processing time of parallel application.

Roundtrip Latency: Measure of latency using MPI is closer to RPC (as shown in Fig. 2). The MPI is based on secure shell, it does not need a permanent daemon as required in PVM and Java RMI. In case of PVM and Java RMI, the daemon is responsible for all communications. It provides flexibility but it eats up the processing power of the machine, due to this reason the PVM and Java RMI take more time to pass the message (high latency).

The Java RMI has higher round trip latency when compared to MPI and PVM as the communication is done in terms of objects. When communication is done over the network using Java RMI, the objects are converted into bytes and then these bytes are transmitted over the network. The conversion of objects into bytes is a time consuming job. For these reasons, the Java RMI has the highest latency among all paradigms.

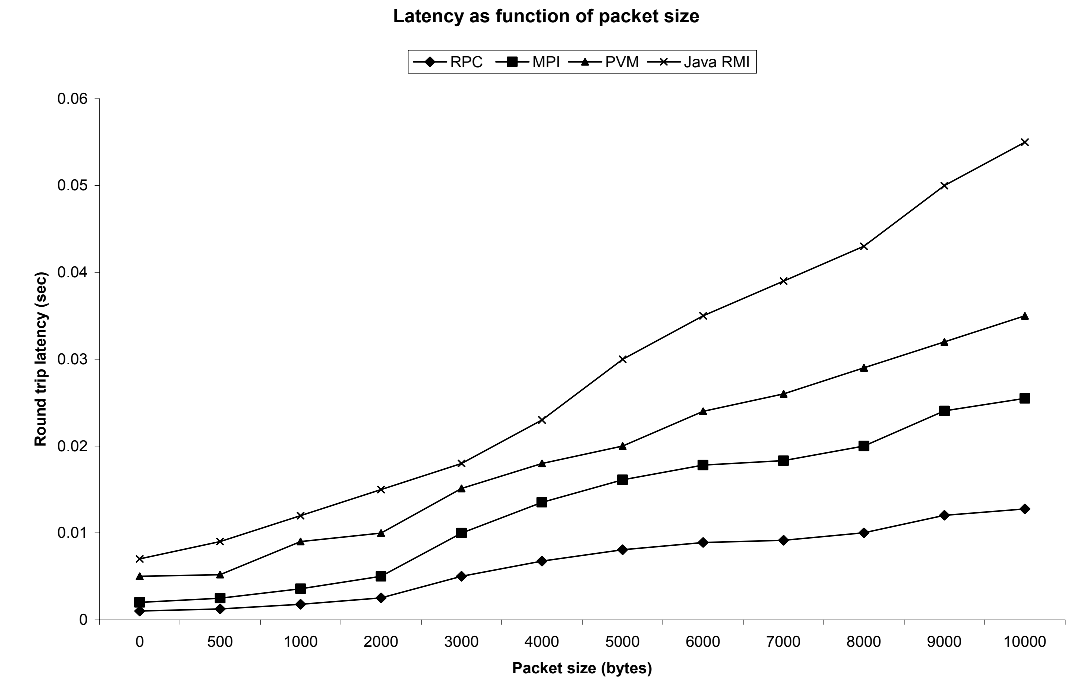

Fig. 2: RPC, MPI, PVM and Java RMI latency measurement.

Bandwidth: The measured bandwidth of all four paradigms is illustrated in Fig. 3 by varying packet size. The average bandwidth measured for RPC is 8.7 Mbps, for MPI is 7 Mbps and PVM is 4 Mbps. Java RMI gives the least performance with respect to bandwidth that is 2.5 Mbps.

Fig. 4 shows the EP benchmark execution times. The result shows that the measure of parallel execution time for MPI is much closer to RPC measure time. The EP parallel measured time for Java RMI is worst in all paradigms. The results show that the execution times of all paradigms decrease with increasing number of PCs.

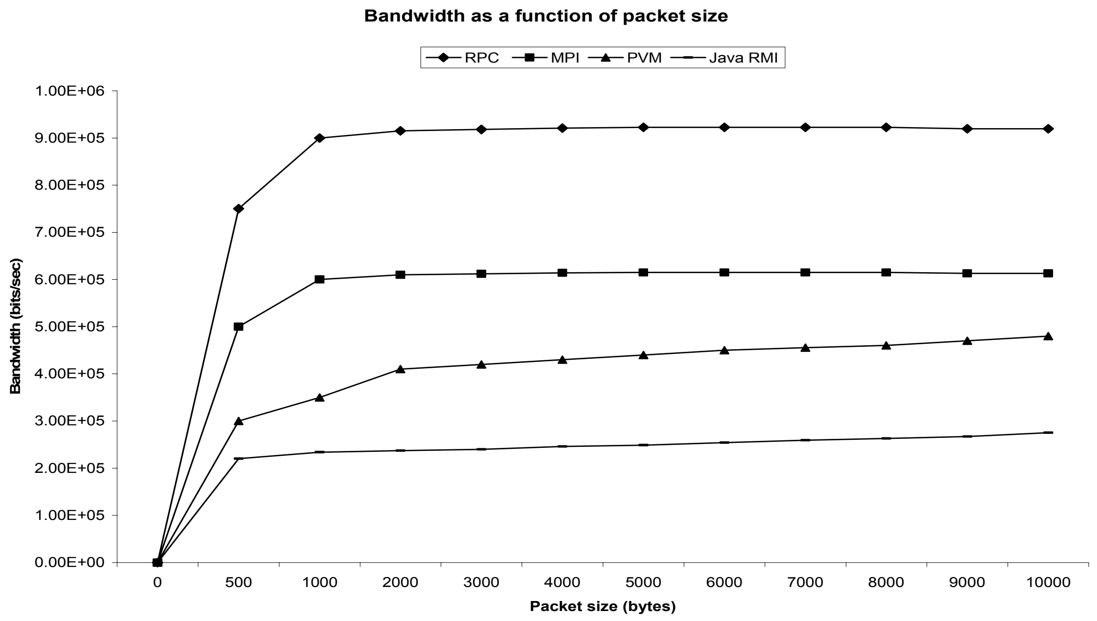

Fig. 3: The measured bandwidth of RPC, MPI, PVM and Java RMI for varying packet size

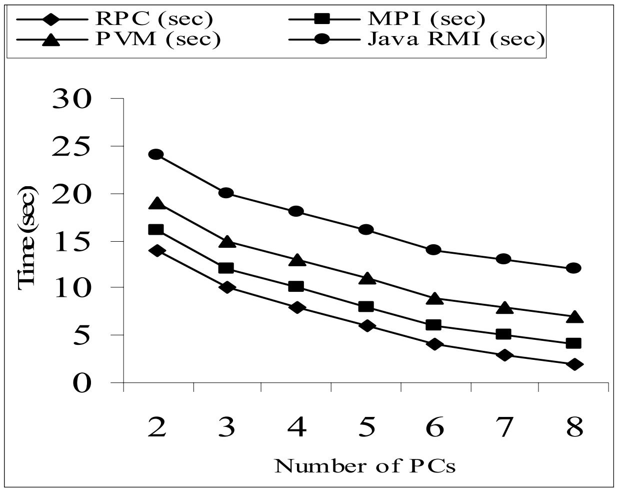

Fig. 4: Results of EP benchmark

5 CONCLUSION

We have practically evaluated the performance of RPC, MPI, PVM and Java RMI. The latency, bandwidth and total processing time were measured using the matrix multiplication application on homogeneous cluster of PCs. RPC was used as a reference tool. The measured results reflect the fact that MPI performance is closest to RPC. PVM performance is 2 to 3 times slower than RPC and Java RMI is 4 to 6 times slower than RPC.

REFERENCES

[1] W.Groop, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd edition, MIT Press, Cambridge, MA, 2002

[2] N. Karonis, B.Toonen, and I. Foster, “MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface”. *Journal of Parallel Distributed Computing*, 2003

[3] MPICH-G2: http://www.hpclab.niu.edu/mpi, 2003

[4] MPI Standard: http://www-unix.mcs.anl.gov/mpi/

[5] V.S. Sunderam, “PVM: A framework for parallel distributed computing”. *Concurrency: Practice and Experience*, vol.2(4), pp.315-339 (1999).

[6] J. Flower and A. Kolawa, “Express is not just Message Passing Systems: Current and Future Direction in Express”, *Parallel Computing. 20* (2003), pp 597-614.

[7] R. Butler and E. Lusk, “Monitors, Message and Clusters: the p4 Parallel Programming System”, *Journal of Parallel Computing*, 20(4), 547-564, April 1999.

[8] Java RMI Documentation : http://java.sun.com/j2se/1.4.2/docs/guide/rmi/

[9] John Bloomer, *Power Programming with RPC*, O’Reilly & Associates, 1998.

[10] *PVM 3.3 User’s guide and reference manual*, Oak Ridge National Lab., Oak Ridge, Tennessee.

[11] Message Passing Interface, MPI Standard: http://www.mpi-form.org.

[12] J. Dongarra and T. Dunigan, “Message-Passing Performance of Various Computers”. *Concurrency: Practice and Experience 9(10),* pp 915–926, 1997, ISSN 1040-3108.

[13] Michal Soch and Pavel Tvrdik, "Performance evaluation of message passing libraries on IBM SP-2": http://cs.felk.cvut.cz/pcg/html/supeur96/

[14] G.A.Gesit, J.A.Kohl, "PVM and MPI: A Comparison of Features": www.csm.ornl.gov/pvm/PVMvsMPI.ps

[15] Adams, Mary, "Java RMI : An Overview", *Proceedings of the IEEE Southeast Con Conference*, pp. 25-30, April 1999

[16] Sanjay P. Ahuja, Renato Quintao, "Performance Evaluation of Java RMI: A Distributed Object Architecture for Internet Based Applications", *8th International Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems*, August, 2000.

[17] Breg, F., Diwan, S., Villacis, J., Balasubramanian, J., Akman, E., and Gannon, D. 1998. "Java RMI performance and object model interoperability: Experiments with Java/HPC++ distributed components". In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing* (Santa Barbara, CA), ACM, New York, NY.

[18] Hirano, S., Yasu, Y., and Igarashi, H. "Performance evaluation of popular distributed object technologies for Java". In *Proceedings of the ACM 2000 Workshop on Java for High-Performance Network Computing.*

[19] R. Z. Khan, A. Q. Ansari and Kalim Qureshi "Performance Prediction for Parallel Scientific Application*", Malaysian Journal of Computer Science, Vol. 17 No. 1, June 2004, pp 65-73.*

BIOGRAPHIES

Kalim Qureshi is an Assistant Professor of department of Computer Science at Kuwait University, Kuwait. He published more than 15 International journal papers. His research interests include network parallel distributed computing; thread programming, concurrent algorithm designing, task scheduling, and performance measurement. Dr. Qureshi is a member of IEE Japan and IEEE Computer Society.

Haroon Rashid is a Director and Faculty member of Department of Computer Science at COMSATS Institute of Information Technology, Abbattabad Campus, Pakistan. His research interests include parallel computing, distributed systems, high speed networks, multimedia, and network performance optimization