Cluster Programming Environments

Scott Needham AND Trevor Hansen

School of Computer Science and Software Engineering
Monash University
Clayton Campus, Melbourne, Australia

Email: {sneedham, hansen}@cs.monash.edu.au

Abstract

A major issue slowing the development of cluster computers is that programs that efficiently take advantage of them are difficult to write. Cluster programming adds a whole facet to sequential programming; requiring specification of not just when an operation will run, but also where it will be ran. Communications packages such as MPI and PVM explicitly express communications between the nodes of the cluster, while object orientated schemes instead focus on transparency, freeing the programmer from explicitly describing the parallelism. There is currently a great divide between the efficiency of both schemes, with MPI/PVM systems dramatically outperforming applications based on Java.

Freeing the programmer from having to explicitly include parallel primitives is enticing indeed and when combined with the interoperability and portability benefits of Java, together mean that Java will soon be a serious choice for a high performance cluster programming environment.

1. Introduction

High performance cluster computing is the growing field of linking together many inexpensive commodity computers and using them to solve difficult problems in commerce (e.g. Web Hosting) and science (e.g. Climate Modeling). Cluster programmers require programming environments that allow them to take advantage of the clusters performance capabilities easily. While there are many hundreds of existing parallel programming languages, each has been designed for a specific architecture and most don't operate well upon cluster computers. Clusters have much higher latency and lower communications throughput than parallel computers, and so languages are required that have as a primary design focus that of limiting I/O- something that is hardly a concern for a shared memory parallel computer. These considerations of data distribution and locality are primary to the success of any cluster programming environment, and immediately rule out the majority of Parallel programming languages as suitable choices. These parallel languages would need at best major revision if they are to become cluster programming environments.

It has been realized that for clusters to be feasible, a parallel programming framework for heterogeneous distributed computing has to be developed. This was the driving force behind the creation of the Parallel Virtual Programming (PVM) environment. This environment gives the user the ability to write code for the PVM virtual computer and PVM takes care of how this code is execute on the computers which make up the cluster on which the code is run.

Further so much of the slow progress in Parallel Systems can be traced back the abundance of proprietary languages and implementations, each manufacturer had their own systems, so changing parallel computers was a significant undertaking requiring large scale software rewriting. Successful cluster programming environments necessarily have broad acceptance, providing vendors the opportunity to market their products (with minor revision) to a broad community of cluster users, so encouraging the development of high quality tools. Efficient, widely used cluster programming environments facilitate the spread of mature cluster tools.

To go some way towards standardization and remedy the diversity in the message passing environments being developed by the different parallel processor vendors, in 1993 a group of high performance computing experts gathered together to define a standard in message passing between nodes in a cluster. As a result the MPI standard was

created which gave the vendors a portable, high performance cluster-programming environment. The MPI API provided a huge number of standard message passing functions, far in excess of any of the existing systems.

Directly programming inter-node communication with sockets is the equivalent of writing sequential programs in assembly code, cluster computing experience and software engineering generally, have shown that efficient program development requires languages that are at a high a level as possible (until performance mandates lower level control). Explicitly programming inter-node communication produces code that while fast, is unreliable and expensive to maintain (like assembly code). But even though the current crop of Java environments require limited code modification (really only the usage of a different compiler) they regrettably perform around 4 times worse for scientific problem solving [6] than a similar FORTRAN application.

The load on each cluster will vary during program execution, as tasks complete and the computation shifts. The programmer is left with the choice of either distributing computation haphazardly, producing poorly-performing programs, or spending more development time including load-balancing code in the application. Several cluster programming languages automate this, including progress migration capabilities , and other load balancing strategies besides those provided by the management system.

A good cluster programming environment is hardly enough to ensure the wide adoption of cluster computers generally, simultaneous increases in network speed, and the development of more advanced cluster monitoring and operating systems are also necessary to continue the growth in cluster based computing.

2. Works in this area

Name	Description	Remarks	URL
Arjuna	An object-oriented system that implicitly handles the majority of the work usually associated with developing parallel applications. Applications operate using a client-server model and can execute in a heterogeneous environment.	No longer under development.	**http://arjuna.ncl.ac.uk/Arjuna/index.htn**
CILK	Inclusion of some extra primitives in the C language. An Algorithmic multithreaded language Runtime system has the responsibility of scheduling the computations.	Down scalable- runs on single process systems without modification	**http://supertech.lcs.mit.edu/cilk/**
HORB	Injects a few commands into Java to enable objects to be spread across a heterogeneous cluster.	100% Java (interoperable), Open source,	**http://horb.etl.go.jp/horb/doc/index**
CVM	Coherent Virtual Machine, automatically load balancing, distributed shared memory, C++ source code freely available.	No longer under development	**http://www.cs.umd.edu/projects/cvm/**
GLU	Granular Lucid Toolkit, features automatic load balancing, target-specific executables,	No longer being developed	**http://www2.csl.sri.com/glu/html/paper**
Linda	Tuple based language-independent	Great idea,	**http://www.cs.egr.edu/linda-group/inde**

	primitives that are inserted into the host language		
MPI	A message passing interface, generally language independent even though bindings are included in the standard for C and FORTRAN	One of the most popular libraries for cluster computing.	**http://www-unix.mcs.anl.gov/mpi/index**
CHARM	Extra primitives are included into C.	Looks like Joyce, but like CILK too	**http://charm.cs.uiuc.edu/**
PC++	C++ extensions that permit member functions to run on all objects simultaneously. A preprocessor transforms pC++ code into C++	Straight forward addition to C++	**http://www.extreme.indiana.edu/sage/**
JavaParty	Transparently adds remote objects to Java, a preprocessor produces straight Java code [10]	Much faster than RMI, less hassle than sockets.	**http://wwwipd.ira.uka.de/JavaParty/**
PVM	permits a heterogeneous collection of machines to be joined together to produce a Parallel Virtual Machine.	Trade some speed for the virtual machine ideal.	**http://www.epm.ornl.gov/pvm/pvm_hor**

3. PVM

The development of PVM started in the summer of 1989 when Vaidy Sunderam, a professor at Emory University, visited Oak Ridge National Laboratory to do research with Al Geist on heterogeneous distributed computing. They needed a framework to explore this new area and so developed the concept of Parallel Virtual Machine. PVM is a software package that permits a heterogeneous collection of machines, from laptops running Windows 95/NT to multiprocessor supercomputers running Unix, to be hooked together in a network and be used as a single large parallel computer. The overall objective of the PVM system is to enable a collection of such computers to be used cooperatively for concurrent or parallel computation. So giving the user the power of a super computer via the linking of a number of computers which is only limited by the quantity accessible by high speed LAN connections and the world wide web. Hundreds of sites around the world are using PVM to solve important scientific, industrial, and medical problems. With tens of thousands of users, PVM has become the de facto standard for distributed computing worldwide.

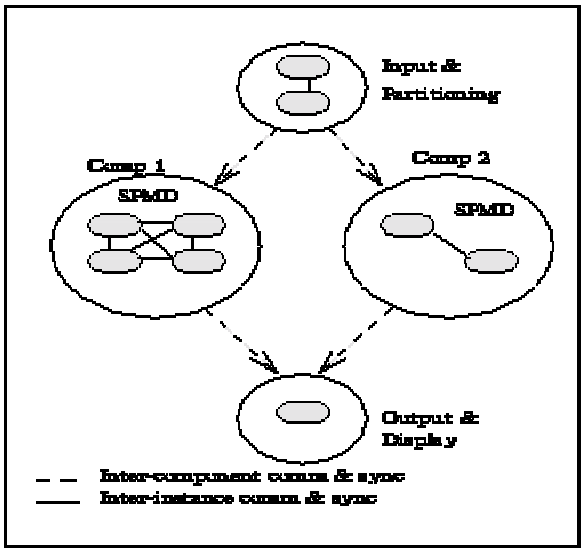
The principles upon which PVM is based include the following:

- User-configured host pool: The application's computational tasks execute on a set of machines that are selected by the user for a given run of the PVM program. Machines can be added or deleted during operation.
- Translucent access to hardware: We can take advantage of the capabilities of certain machines in the pool by using then for specific tasks, or the hardware environment can be viewed as a collection of equal virtual processing elements.
- Process-based computation: The task is PVM's unit of parallelism and is an independent sequential thread of control, which partakes in both communication and computation.
- Explicit message-passing model: This collection of tasks cooperate by explicitly sending and receiving

messages to one another. There is no language dependant restriction on the size of any message.

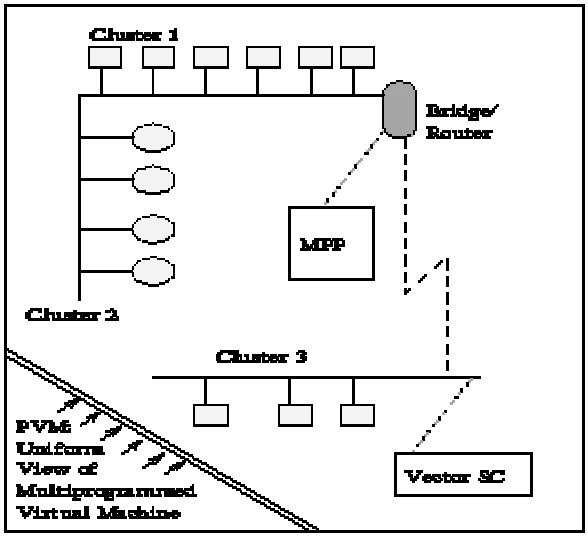
- Heterogeneity support: The PVM system supports heterogeneity in terms of machines, networks, and applications. With regard to message passing, PVM permits messages containing more than one datatype to be exchanged between machines having different data representations.
- Multiprocessor support: PVM uses the native message-passing facilities on multiprocessors to take advantage of the underlying hardware.

The PVM system is composed of two parts. The first part is a daemon, which resides on all the computers making up the virtual machine. It is often referred to as pvmd and is designed so any user can install this daemon on a machine. A virtual machine can be created by starting up PVM, after which PVM programs can be executed. Each user has the capability to execute several PVM applications simultaneously.



(a) PVM Computation Model

The second part of the system is a library of PVM interface routines, consisting of a repertoire of primitives that are needed for cooperation between tasks of an application. The library provides routines for message passing, spawning processes, coordinating tasks, and modifying the virtual machine.



(b) PVM Architectural Overview

With PVM being the leader in the rapidly growing area of heterogeneous distributed computing it's not surprising that we have seen a flood of PVM extensions and tools aimed at improving the capabilities and power of the PVM environment. Below we will discuss some of the more prominent examples:

XPVM

XPVM provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. These views can be used to assist in debugging and performance tuning by providing information about the interactions among tasks in a parallel PVM program, as well as information on machine usage. If we have compiled a PVM program to capture tracing information at run-time, any task spawned from XPVM will return trace event information for analysis in real time or for post-mortem playback from saved trace files.

CPPVM & PVM++

CPPvm is an extension of PVM that allows processes to pass C++ objects to each other. CPPvm has some standard message passing classes, which provide functions necessary to write cluster programs, although for more sophisticated problems user defined message passing classes must be development.

For a C++ extension of PVM to become successful the wide spread acceptance of such a language would be required. Then we might see the emergence of a standard message passing template library for cluster programming applications. Even standard parallel sorting, searching, etc classes giving the user a far more powerful environment allowing for rapid prototyping. PVM++ is another C++-library for PVM also focused on combing C++ into the PVM environment.

jPVM & JPVM

jPVM extends the capabilities of PVM to the world of Java™, Sun Microsystems Inc.'s architecture-independent programming language for the Internet. jPVM allows Java applications and applets as well as existing C, C++, and Fortran applications to communicate with one another using the PVM API. This means that Java programs can be built to interface to existing C, C++, and Fortran programs and use PVM to ship data from those programs to the Java interface. Or you could use this as a communications package as you transfer applications from C or C++ to Java.

jPVM is not an implementation of PVM written in Java TM. There is a package called JPVM, that is a 'PVM-like library of object classes' for parallel programming in JavaTM.

Some Other Extensions of PVM...

- Parallel plug-in interface that allows users or applications to dynamically customize, adapt, and extend the environments features.
- There are a huge number of extensions of PVM that provide a more powerful application of the PVM concept. Here we will list just a few,
- PVM Toolbox for Matlab: Javier Baldomero has created a toolkit for calling PVM from Matlab.
- Perl-PVM, Pypvm: Perl and Python extensions for PVM.
- HP-PVM, Compaq-PVM: Fast commercial versions of PVM.

HARNESS

Looking to the future, Harness builds on the concept of the Distributed Virtual Programming environment, but fundamentally recreates this idea and explores dynamic capabilities beyond what PVM can supply. The Harness project is focused on developing three key capabilities within the framework of a heterogeneous distributed computing environment,

- Parallel plug-in interface that allows users or applications to dynamically customize, adapt, and extend the environments features.
- Distributed peer-to-peer control that prevents single point of failure (in contrast to typical client/server control schemes). Greatly enhances the fault tolerance that is available to large, long running simulations.
- Multiple distributed virtual machines that can collaborate, merge, or split. This feature provides a framework for collaborative simulations.

4. MPI

In contrast to the PVM API, which sprang from and continues to evolve inside a research project, the MPI was specified by a committee of about 40 high performance computing experts from research and industry in a series of meetings in 1993-1994. MPI was designed to do away with the growing number of machine dependant message-passing APIs being designed by the Massively Parallel Processor (MPP) vendors. MPI is intended to be a standard message-passing specification that each MPP vendor would implement on their system, making it possible to write portable parallel applications. MPP vendors need to be able to deliver high performance which thus became a focus in the design of the MPI API. This would be essential if MPP vendors were to accept the end product and in fact the API became standard in practice. Given this design focus, MPI is expected to always be faster than PVM on MPP hosts. MPI-1 contains the following main features:

- A large set of point-to-point communication routines (richer than any other).
- 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 datatypes that describe messages of noncontiguous data.
- Because of portability issues across a network of in 1995 the MPI committee began meeting to design the MPI-2 which was enhanced to include:
- MPI SPAWN functions to start both MPI and non-MPI processes.
- One-sided communication functions such as put and get.
- Non-blocking collective communication functions.
- Language bindings for C++.

Some Differences Between MPI and PVM

PVM fully implements process control, that is the ability to start and stop tasks, to find out which tasks are running, and possibly where they are running. On the other hand MPI-1 has no method to start a parallel program, MPI-2 is being designed to handle these problems [7].

PVM's inherently dynamic nature gives it the power to do resource management. Computing resources, or "hosts", can be added or deleted at will, either from a system "console" or even from within the user's application. MPI lacks such dynamics and is, in fact, specifically designed to be static in nature to improve performance.

In MPI a group of tasks can be arranged in a specific logical interconnection topology. The topology with which the tasks communicate within should then be mapped to the underlying physical network topology resulting increased speed for message transfers. PVM does not support such an abstraction.

PVM has supports a basic fault notification scheme which is under the control of the user. Tasks can register with PVM to be "notified" when the status of the virtual machine changes. This notification comes in the form of special event messages that contain information about the particular event. The current MPI standard does not include any mechanisms for fault tolerance, although MPI-2 is being designed to incorporate fault tolerance similar to that of PVMs.

One of the major differences between the two environments is the MPI communicator. The communicator can be thought of as a binding of a communication context to a group of processes. Having a communication context allows library packages written in message passing systems to protect or mark their messages so that they are not received by the user's code.

PVMPI

At University of Tennessee and Oak Ridge National Laboratory an investigation has recently begun looking at the possibility of combining the features of both MPI and PVM to create PVMPI. This will result in an environment with virtual machine features of PVM and the advanced message passing features of MPI. The enhanced environment will now include PVM's access to virtual machine resource control and fault tolerance and would use PVM's network communication to transfer data between different vendor's MPI implementations allowing them to interoperate within the larger virtual machine.

5.HORB

HORB is a Java based distributed object system, an ORB (Object Request Broker) that enables object oriented computing; it can even be ran across clusters or distributed networks that aren't necessarily running the same Operating System. Methods are called via proxy objects which contain program stubs that make requests to the ORB for methods to be called up the actual object. Java is a multi-threaded single processor language but it can only be ran on a single machine, HORB allows us to spread the execution over many different machines and many types of machines.

An Object Request Broker is a layer that manages requests that are made to objects that are not located on the current computer, other ORBs besides HORB exist the most noteworthy being DCOM, and the OMG's CORBA. These are both general ORBs while HORB is specifically designed for Java

Due to HORB's implementation being 100% Java, and it being a primary design consideration of Java to achieve interoperability. HORB has the advantage over other Cluster programming environments that the cluster need not even be running the same Operating System. The Java programs ran on the cluster do not even need to be recompiled when used on a heterogeneous cluster- a great advantage when varied computers comprise the cluster. Many programming environments offer source level interoperability, but binary level interoperability is one of the great advantages of HORB and other Java based cluster programming environments. The Java basis is another strong advantage it is enough to remember the lists of computers that other Cluster Programming Environments support- Linux, Solaris, AIX,. But since HORB is Java based it doesn't require a port, just a virtual machine!.

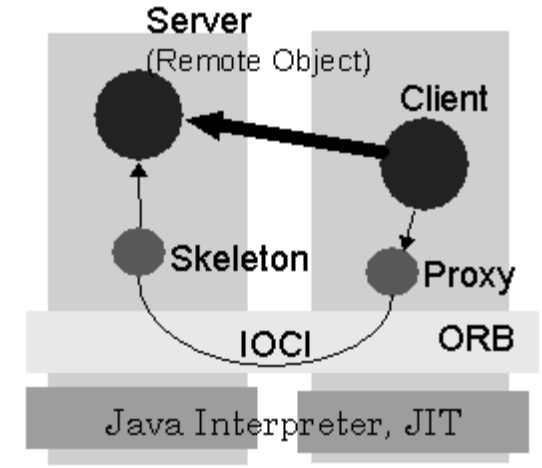
It can't be stated enough the advantages of HORB and similar ORB schemes, they take normal Java code and enable it to be parallelised across a cluster with at most trivial changes to the source code. Admittedly a strong C-FORTRAN implementation will easily outperform a Java-HORB one (see bellow for examples) but just as today it takes an expert assembly programmer to outperform a C-compiler, soon the day will come when only with great

care a C-FORTRAN will outperform an automatically generated ORB system.

HORB consists of the HORBC compiler which uses a non-modified javac compiler as the back-end, and can compile unmodified Java programs into distributed objects. The HORB server which is the object request broker, and the HORB class library. It works with the Javac compiler, interpreter and system classes distributed by Sun, but none of Sun's code needs to be modified for HORB.

Many cluster/parallel programming environments have been created- many are listed in the above table that are no longer being developed, of interest is that the systems that have maintained viability are those that inject primitives into languages. New parallel languages have been proposed that better enable the programmer to express inherent parallelism, but rather than completely new parallel languages, programmers prefer slight additions to currently known languages . This is a great strength of HORB! as shown below it injects only half a dozen new primitives into Java all of which are very common sense, this feature in no way ensures popularity, Linda for example adds very few primitives, but remains little known. A common failing of other Java Distributed object systems is that they require distributed objects to either have a specific interface(e.g. remote for JavaRMI) or inherit special "distributed classes" because Multiple inheritance is not allowed in Java this causes grievous limits on the inheritance hierarchy, in some cases requiring considerable redesign, HORB has the advantage that no "distributed class" needs to be inherited!.

An advantage of HORB is type safety and that polymorphism is maintained. The objects can be casted and treated just like normal, providing much freedom to the programmer. Further it uses Global Garbage Collection, reference counting keeps track of the current number of references to an object, and when it is no longer referred to it is destroyed, just like normal!



Model of HORB structure[10]

In this example a client object whishes to take advantage of the methods of an object housed on a different computer, it accesses these methods through a proxy class. Which directs the request to the Object request broker, given identification on which object is required it locates the actual object and via a skeleton object calls the method upon that. The proxy in this case behaves exactly like the actual object, appearing to the client as if it is actually the object. The proxy and skeleton object are automatically generated by the HORBC compiler, so the programmer is freed from the task of explicitly including the necessary parallelism.

This is scheme best describes functional rather than domain decomposition, In the functional decomposition approach the focus is on dividing the computation into disjoint tasks. An example of a modern problem for which functional decomposition is most appropriate is those where the problem involves a number of simpler models connected through interfaces. For examples, a simulation of the earth's climate may comprise components representing land, air, ocean, would benefit immensely from a model like HORB.

```
class Server{ // This server class is standard Java compiled by HORBC
String greetings (String name)
```

```
{ Return "Hello," + name;}}

 Class Client{
Public static void main (string[]) {
Server_Proxy server = new Server_Proxy("horb://www.etl.go.jp:8887/");
System.Out.println(server.greetings("World!")); }}
```

Example Java Code using HORB [2]

The "server" code is normal Java code, the client object that is making a call upon the server creates a proxy class to the object (new Server_Proxy). In this example the actual object is created on the distant computer www.etl.go.jp, the next line shows the remote object being used via the proxy. It should be noted that the object may or may not be actually being created on the remote computer when the new command is issued, firstly the client needs to have the required access, secondly unlike CORBA and JavaRMI, HORB supports both object connection and object creation. In the object connection model all of the clients share one remote object, and the instance variables are either stored in special data storage or transmitted on each call. In the object creation model a new object is created on each "new" this a remote object will have the client's specific data Since one objects are owned . HORB allows both schemes while CORBA and JavaRMI are limited to the object creation model.

This example illustrates both the major ideas and failing of HORB it is really designed as a client-server communication model, many people believe that the cluster is actually a peer group, and that models such as HORB which emphasize a client-server nature are ill-conceived.

The Object Management Group's (OMG) CORBA(Common Object Request Broker Architecture) is another example of an object broker, but CORBA is intended to realize interoperability among languages. To achieve this it uses IDL files, for each class that is defined a new IDL file needs to be written. HORB being specifically a Java system has no such constraints, and automatically generates the required proxy and skeleton classes and so requires less of the programmer and so provides less burdensome application development.

An excellent performance evaluation is detailed in Hirano[10] which compares the then HORB implementation to a socket implementation and javaRMI. It finds that HORB is significantly faster than both other schemes. Further in their comparison of JavaParty with RMI and Sockets, Philippsen et. Al. [10] find that RMI and sockets require significantly more effort to implement than a corresponding ORB implementation.

We run our programs on high performance clusters inorder to speed execution, but Java is often accused of being too slow for serious programming, especially for scientific problem solving. In their study of the speed of a Java Vs. FORTRAN geophysical modeling system, they find the Java code compiled with the JIT compiler and using RMI was 4 times slower than the equivalent. But promising research on object serialization [5] coupled with ongoing development is reducing this difference dramatically.

6. Summary and Conclusions

PVM is an environment that allows a programmer to use a single virtual machine as an interface to a cluster of possibly heterogeneous machines, while not as efficient as MPI it provides a more intuitive programming environment, so speeding program development. The MPI standard is a collection of communication routines that is inserted into a language to allow the explicit specification of inter-node communication, currently MPI though has the failing of, unlike PVM, not allowing dynamic process creation, but the latest addition to the standard MPI-2 includes this facility. Many programmers are turning to the newly defined MPI standard, because of its enhanced message passing facilities, although they are reluctant to let go of some of the superior PVM features. The MPI2 standard currently in development is set to remedy some of these inadequacies and could see PVM becoming obsolete.

HORB and other object based cluster paradigms share more with PVM than MPI, they usually require only small changes to generate the cluster algorithm, while MPI requires the explicit specification of the inter node communication and the allocation of tasks. HORB provides an extension to the popular Java programming environment, the HORB language is valuable because it requires the user to have virtually no knowledge of the under lying parallel system. Java's object orientated approach allows rapid program development, encourages software reuse, and is

binary rather than source interoperable- meaning that source doesn't need to be recompiled when a heterogeneous cluster is being used. Currently HORB and other similar systems like the JavaRMI are slower than equivalent FORTRAN/MPI systems but their maintainability, and ease of construction means that as performance improves they will become only more used.

MPI provides an extremely efficient basis for an application, partly because of its large range of message passing functions, but also because of its restrictions on expensive operations, like dynamic task creation. MPI is the library of choice for performance critical applications.

HORB/ PVM and other systems that abstract away the cluster's detail , are ideal for rapid program development, programs built using these systems will not have the performance of their MPI counterparts, but this may be compensated for by their ease of construction.

The success of Cluster Programming environments is seen in the wide spread use of cluster computers, nowadays the cutting edge Web Servers, Scientific Modeling Systems, and Data Warehouses problems are all implemented on cluster computers.

There is a bright future indeed for cluster computing, the huge demand for computer power coupled with the inexpensive of commodity components means that cluster computing will only become more prevalent. The development of new generation languages and other cluster tools means that program development will become easier and more straightforward.

References

[1] HORB users guide: http://horb.etl.go.jp/horb/doc/guide/intro.htm

[2] Hirano, Satoshi, *HORB: Distributed Execution of Java Programs*, pp29-42 Proceedings of WWCB '97 LNCS, V1274 Springer, Berlin '97

[3] Philippsen, M & Zenger M. Java Party- Transparent Remote Objects in Java, Concurrenty: Pratice & Experience, Vlume 9, Number 11, pp. 1225-1242, November 97

[4] Phillippsend, Imperative concurrent object-oriented languages. Techinical Report TR-95-050, Internation Computer Science Institute . Berkeley, August 1995.

[5] Philippsen,B et. al. More Efficient Object Serialization. Parallel and Distributed Processing, LNCS 1586 pp718-732, International Workshop on Java for parallel and distributed computing, San Juan. Puerto Rico. April 12, 1999

[6] Jacob, M. et al. Large-Scale Geophysical Algorithms in Java: A Feasibility Study Concurrency: Practice and Experience, 10(11-13): 1143-1154 September-November 1998.

[7] G. A. Geist, J. A. Kohl, P. M. Papadopoulos, PVM and MPI: a comparison of Features, G. A. Geist. J. A. Kohl.