A Comparative Study of Parallel and Distributed Java Projects for Heterogeneous Systems

Jameela Al-Jaroodi, Nader Mohamed, Hong Jiang, and David Swanson
Department of Computer Science and Engineering
University of Nebraska – Lincoln
Lincoln, NE 68588-0115, [jaljaroo, nmohamed, jiang, dswanson @cse.unl.edu]

Abstract

During the last few years, the concepts of cluster computing and heterogeneous networked systems have received increasing interest. The popularity of using Java for developing parallel and distributed applications that run on heterogeneous distributed systems has also grown rapidly. This paper is a survey of the current projects in parallel and distributed Java. These projects' main common objective is to utilize the available heterogeneous systems to provide high performance computing using Java. These projects were studied, compared and classified based on the approaches used. The study shows three major approaches. One is to develop a system that replaces the Java virtual machine (JVM) or utilizes the available parallel infrastructure such as MPI or PVM. Another is to provide seamless parallelization of multi-threaded applications. The third is to provide a pure Java implementation by adding classes and features that support parallel Java programming. In addition, a number of open issues are identified and discussed in this paper.

1. Introduction

The need for high performance and powerful computing resources led to great advances in the area of distributed and parallel computing. Currently cluster machines, grids and heterogeneous networked systems can provide processing power comparable to special purpose multi-processor systems for a fraction of the cost. This direction had also led to the need for system and application software that can support such systems and provide the user with transparent and efficient utilization of the multiple machines used in the cluster or distributed system.

This paper studies projects that involve providing parallel and distributed processing facilities in Java for clusters and heterogeneous networked systems. These projects were compared in terms of the approach used, the compatibility with JVM, the level and granularity of parallelism and other features. Also some quantitative comparisons were made wherever possible to show the relative performances of such projects.

This study is meant to provide researchers and practitioners with an overview of the status of research in parallel and distributed Java. It also identifies some of the problems and open issues in this area of research. The paper provides some background information in section 2. Section 3 reviews the projects. A discussion of the projects studied is presented in Section 4 which also identifies a number of open issues in the area, while Section 5 concludes the study.

2. Background

This section presents a brief introduction to some concepts related to the study such as RMI, serialization, reflection and MPI. Many of the features and standards discussed were utilized, and in some cases improved or modified, by different research groups to develop their projects.

2.1 Java Remote Method Invocation (RMI)

Sockets, in Java, provide programmers with the flexibility to write efficient distributed applications, but they tend to make the development process more complex. Remote Method Invocation (RMI) [23] was introduced as an alternative to socket programming. It creates a layer that hides the details of communications to the level of procedure call (method invocation) from the developer. RMI simplifies the development process, but it reduces the efficiency of the application. Many projects utilize RMI as a communication mechanism between the participating nodes.

2.2 Reflection API

The reflection API represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine. With the reflection API, it is possible to determine the class of an object and get information about a class's modifiers, fields, methods, constructors, and superclasses. It is also possible to find out what constants and method declarations belong to an interface, create an instance of a class whose name is not known until runtime and get and set the value of an object's field, even if the field name is unknown to the program until runtime. In addition, one can invoke a method on an object, even if the method is not known until runtime, and create a new array, whose size and component type are not known until runtime, and then modify the array's components.

2.3 Serialization

One of the new features added to Java 2 is the public interface Serializable [24]. When a class is serializable, any object instantiated from that class can be stored or sent to any remote machine. When an object is serialized, it is converted into a byte stream that contains all the required information to reconstruct that object again. Many projects use serialization to provide shared objects or object-exchange mechanisms.

2.4 Class loaders

Java class loader is responsible for loading the Java classes (bytecode) on a JVM. Java allows programmers to override the default class loader by writing their own method for class loading. This gives the programmer the flexibility to add features to the class loader to achieve different functionality at run time.

2.5 MPI

Message Passing Interface is a library of routines provided for users who wish to write parallel and distributed programs. MPI-1 was developed for use mainly with FORTRAN and C language. It provides a number of library functions to exchange messages between processes. Later MPI-1.2 and MPI-2 were developed as extensions of MPI-1, adding more functionality such as process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings such as C++ bindings.

Object-Oriented MPI was introduced a few years ago with the main objective of providing C++ programmers with more abstract message-passing methods. A number of extensions were developed to provide object orientation for C++ and FORTRAN 90 such as OOMPI [16, 21], Charm++ [14] and ABC++ [17].

Later with the success of Java as a programming language for regular and distributed applications, some effort was made to provide extensions of MPI that can be used in Java. The Java Grande Forum has developed a draft standard for message-passing in Java (MPJ) [9] that was based on the well-known MPI standard. Many research groups used this as a basis for implementing a parallel Java environment using message passing. One example is mpijava [6], which provides a Java API closely similar to C++ bindings in MPI-2. This approach requires users to have the MPI library, a Version of Java and a C compiler in order to link the MPI library with the mpijava classes via the Java native interface (JNI). Another example is jmpi [10], which is also built based on the MPI-2 C++ bindings, but it requires a Java to C interface (JCI) that compiles the Java programs into C before being able to run them.

3. projects

In this section a number of projects are discussed. Many of these projects are in a research phase.

3.1 Titanium [25]

Developed at the University of California Berkeley, Titanium is a Java dialect used for large scale scientific computing. It provides parallelization primitives in a Java-like language. It provides immutable classes, flexible and efficient multi-dimensional arrays, an explicitly parallel SPMD model, distributed data structures, and zone-based memory management. One advantage is that programs written for a shared memory model can be run without modification on a distributed system. Titanium compiler compiles Titanium programs into C. Titanium is not compatible with JVM; however, it inherits some of the safety features of Java.

3.2 Waterloo University Research Projects

A series of projects: ParaWeb [7], Ajents [13], and Babylon [12] were developed at York and Waterloo University.

ParaWeb [7] allows users to utilize the Internet computing resources seamlessly. It allows users to upload and execute programs on multiple machines on a heterogeneous system. Using ParaWeb clients can download and execute a single Java application in parallel, on a network of workstations or they can automatically upload and execute programs on remote compute servers. ParaWeb has two implementations:

1. Java Parallel Class Library (JPCL): It facilitates remote creation and execution of threads. It provides communication using message-passing through send and receive primitives. This implementation allows users to write their own parallel applications using the JPCL.
2. Java Parallel Runtime System (JPRS): in this implementation, the Java interpreter is modified to provide the illusion of a global shared address space for the multi-threaded application. This approach requires no changes to the Java programs or Java byte-code.

Some test results are available on ParaWeb using matrix multiplication (400 x 400 integers) on SUN SparcStation-10's 143MHz CPU and 64 MB RAM, connected by 10Mbps Ethernet. The speedup obtained was 3.8 for four processors and 7.4 for eight.

Ajents [13] is a collection of Java classes and servers, written in standard Java, that provide seamless implementation of distributed and parallel Java applications. It requires no modifications to Java language or the JVM. An Ajents server is installed on participating servers to allow users to use their computing resources, while Java security features are used to protect these servers.

Ajents provides many features that facilitate the development of distributed Java applications such as: Remote object creation, remote class loading, asynchronous RMI, object migration, and checkpointing, rollback and restart of objects. Object migration creates a relatively high overhead due to checkpointing and serialization/deserialization of the object. Checkpointing consistency is also a problem. This environment is mostly suitable for coarse grain simple applications.

Babylon [12], a Java based system to support object distribution. It inherited many of the Ajents' features, and also has additional features. It allows one to easily create objects, migrate objects at any time (even at execution), seamlessly handle arrival and departure of compute servers and provide I/O through the originating machine. It provides immediate and delayed object migration and checkpointing, rollback and restart of objects. In addition, Babylon uses a basic scheduling server.

Babylon was tested using integer matrix multiplication on SUN SparcStation-10's 143MHz CPU and 64 MB RAM, connected by 10Mbps Ethernet. Two settings were used with one and eight processors. In this experiment, the master process resides on an external node other than the one or eight servers respectively. The results show a speedup gain of 6.7 for matrix size 500 and 7.3 for 640.

3.3 JPVM [11]

Java Parallel Virtual Machine is a PVM-like library of object classes implemented in and for use with the Java Programming language. The main goal is to provide a system to utilize the available computing resources in a heterogeneous system. JPVM is written purely in Java to achieve portability. It provides explicit message-passing based distributed memory MIMD parallel programming in Java. However, programs written for JPVM cannot be ported to JVM.

Experiments were conducted to measure the overhead of creating tasks and communications. The basic results, according to the JPVM author, were not encouraging. The task creation and communication overhead is high, which means that JPVM is most suitable for coarse grain parallelization. A matrix multiplication (512 x 512) program was used to measure the speedup achieved using JPVM compared to sequential Java program. Speedup reported was 6.7 for nine processors.

3.4 UIUC project [18]

A research group at University of Illinois at Urbana-Champaign is working on a prototype extension for Java to provide dynamic creation of remote objects with load balancing, and object groups [18]. The language constructs are based on those of Charm++ [14]. The parallel Java extension is implemented using the Converse interoperability framework, which makes it possible to integrate parallel libraries written in Java with modules in other parallel languages in a single application. This implementation, using Converse, allows incremental adoption of parallel Java. Existing libraries written in C and MPI, Charm++, PVM, etc. can be utilized in a new application, with new modules written in Java. The Java user code is written in normal Java with the addition of the calls to the new runtime library that provides the parallelization capabilities. The user will also need to write interface file (in CORBA-IDL style) that will be translated by an interface translator provided with the system.

To achieve parallelism, proxy objects and serialization are utilized in addition to asynchronous remote method invocation. The main implementation goals of this system are to minimize native code, maximize the utilization of Java features, and minimize copying as much as possible. This system is designed for multi-lingual parallel programming, and requires using the provided library and creating special interface files. It also requires JNI to interface with converse messaging layer. This system has its own message passing interface and provides a virtual global object space.

3.5 ProActive PDC [22]

ProActive includes comprehensive library for parallel, distributed, and concurrent programming in a MIMD parallel programming model. ProActive provides a metacomputing framework to convert an object to a thread running in a defined remote address space. Objects are classified into passive objects (non-thread objects) and active object (thread objects). In the ProActive framework, a passive object can be activated as thread object running on another node. All method invocations to any of the methods in the active object will be transparently transferred to the node where the object is running. Moreover, the result of invocation of any method will be transparently returned to caller address space.

A sequential Java program can be converted by the user to multithreaded or distributed program by converting some of the passive objects to active objects running as threads in the same or different address spaces. This conversion requires the user to use a ProActive API in the sequential program to define passive objects as active objects. The rest of the sequential code will require no changes. Remote method invocation is a transparently asynchronous call. The main thread can continue executing its code without waiting for the result. The invocation of active object method immediately returns a future object. This object is a reference to where the result of the not-yet-performed method invocation will be placed. The caller thread will be suspended when it reaches a point where it needs to use the result that will be returned by one of the previously invoked remote methods. This is called wait-by-necessity. The wait-by-necessity is a data-driven synchronization mechanism, as opposed to control-driven synchronization among the distributed threads. ProActive utilizes the Java Reflection API and RMI.

3.6 JavaParty [15, 20]

JavaParty provides facilities for transparent remote objects in Java. It extends the Java distributed capabilities by allowing easy porting of multi-threaded Java programs to distributed environments such as clusters. Remote classes and their instances are made visible and accessible anywhere in the distributed JavaParty environment. The JavaParty environment can be viewed as a Java virtual machine that is distributed over several computers, but a trade-off between location transparency and performance considerations becomes inevitable.

Object migration is one way of adapting the distribution layout to changing locality requirements of the application. In JavaParty, objects that are not declared as residents can migrate from one node to another. This is important even if the remote object location does not influence the semantics of the JavaParty program. JavaParty extends the Java language with only one new modifier called remote. A class or thread defined with the remote modifier is declared to be a JavaParty remote class or thread. The fields and methods of a remote object instantiated from remote class can be accessed transparently like in regular Java that works in a single address space. The programmer does not need to allocate objects and remote threads to specific machines; the JavaParty environment deals with locality and communication optimizations.

The JavaParty environment consists of a pre-processor generating regular Java and a runtime system. The pre-processor is used to translate the JavaParty source program to Java code and RMI hooks to add distribution and communication mechanisms among remote objects and threads. The runtime system is a set of components distributed on all the nodes with a central component, called RuntimeManager. The manager maintains the locations of all contributing nodes and the location of all objects. This information is replicated in the nodes to reduce the manager's load. To reduce the access latency to a remote object, different optimization efforts were attempted, including more efficient object serialization and optimized RMI (KaRMI).

3.7 IBM's cJVM [2, 3, 4, 5, 19]

cJVM is a clustered Java virtual machine that will allow multi-threaded applications to run on the multiple nodes of a cluster. The main objective is to allow existing multi-threaded server applications to be executed in a distributed fashion without the need for rewriting them. cJVM creates a single system image (SSI) of the traditional JVM to transparently exploit the power of a cluster. It is an object-oriented model that can make use of the possibility to have remote and consistent replicated objects on different nodes. cJVM is designed to be aware of the underlying cluster and at the same time hides this fact from the application.

The shared object model was implemented by having a master object (the original object defined by the programmer) and proxies. Proxy objects, located on other nodes, are created by the cJVM run time environment to provide mechanism for other threads located on different nodes to remotely access the master object in a transparent way.

Different optimization techniques to reduce the amount of communication among the nodes were deployed. All these techniques enhance data locality by using cashing based on locality of execution and object migration. Caching is achieved at multiple levels including classes, entire objects, and individual fields. Optimization takes advantage of the Java semantics and

common pattern usage. For example, different types of objects and data elements have different types of optimization techniques. In addition, to enhance data locality, the master copy of an object is placed where it will be used not where it was created. The cJVM run time environment can correct false speculations. cJVM is a completely new Java virtual machine that replaces the standard JVM. Experiments were reported using different benchmark applications (N-body, TSP and pBob). In general around 80 percent efficiency was obtained using four processors.

3.8 Agent-Based Parallel Java [1]

In this project, being developed at University of Nebraska-Lincoln (UNL), an environment is built to provide parallel programming capabilities in Java for heterogeneous systems. For example, Java parallel programs written for this system can run on homogeneous multi-processor systems or on a heterogeneous system like a cluster, a collection of clusters or even through the Internet. The system is written completely in standard Java and can be used on any machine that has a Java virtual machine (JVM), which makes it possible to utilize different machines of varying architectures to execute the user program.

Software agents were written to provide the facilities to coordinate and manage the parallel processes and schedule multiple user jobs among the available processors. Each participating machine has an agent on it, responsible for all the resources on that machine. The agents help deploy and run the user processes on the processors available on that machine. The user programs are converted into threads and executed from memory on the remote machines. This approach reduces the overhead of accessing the hard disk, consumes fewer resources for the threads, and enhances the system security. Currently the user is provided an Auto scheduling option in which agents schedule processes for the user. With agents deployed on different machines user parallel programs can execute on any collection of processors on different platforms. These agents also provide a lot of flexibility to expand the capabilities of this environment. For example the agents can provide dynamic load balancing and fault tolerance.

The Java object-passing interface (JOPI) provides the user with an interface very similar to standard MPI and the MPJ interface, but it also exploits the object oriented nature of Java to simplify the process of writing parallel programs. Compared to the standard MPI, JOPI allows the programmer to exchange objects instead of predefined data elements, which simplifies the interface primitives. JOPI was built using the socket programming classes available in Java. Although it complicates the development process, it provides full control over the system and high flexibility in modifying the underlying design to further improve and optimize the performance. In addition, the system becomes faster and more efficient compared to using RMI.

In this system, a client is created that the user will use to run the parallel programs. This client will communicate with the agent(s) to load the program and run it. Matrix multiplication was used to measure the system performance (see table 1). The matrix contains 1440x1440 floating numbers and the program was run on a cluster containing eight dual-processor nodes (800MHz, 512 MB RAM) connected via a 100Mbps Ethernet.

# Procs	1	2	4	6	8	10
Time	**324.23**	158.18	83.29	56.58	42.30	35.52
Speedup	**1.0000**	2.0498	3.893	5.730	7.665	9.128

Table 1. The results of the parallel matrix multiplication (time is in seconds).

4. Discussion

Many research groups are working on providing a parallel and distributed environment for Java. Most of them target clusters and networks of workstations because Java is machine independent. The projects selected are representative of the different approaches identified in this area of research. Each of these projects has its unique features, advantages and disadvantages (see table 2).

4.1 Comparison and classification

Based on the techniques used and the level of user involvement, it is possible to classify the projects into three different groups:

1. Building a new system that replaces the standard JVM or uses the current available infrastructure for parallelism. Examples are Titanium, which compiles Java dialect to C, JPVM that creates a new JVM for parallel processing, ParaWeb (the JPRS implementation), and the project at UIUC, which is based on Converse. This approach gives the development team the freedom to provide many methodologies for parallelism without having to conform to the current standards or JVMs. In addition, it gives the development team the ability to provide the user with different levels of involvement in the parallelization process. This approach may also lead to a more efficient implementation of a parallel and distributed environment. On the other hand, this approach has some disadvantages. First, the system requires different implementations for every different platform used to support heterogeneity. In addition, any enhancements or changes in the standard JVM cannot be easily incorporated in the new system.

Moreover, adding more servers or machines to the system is not trivial.

2. Extending Java with class libraries to provide explicit parallelization functions such as ParaWeb (the JPCL implementation), Ajents, Babylon and JOPI. In this approach, the parallelization is achieved by adding class libraries to extend standard Java. All implementations require some form of demon running on the participating machines. JOPI is the only system that utilizes software agents for this purpose. The main advantage of this approach is that the system will run on any machine that has a JVM, which provides a good support for heterogeneity. Systems using this approach are portable and more scalable since it is easier to add more machines to the system. One disadvantage is that the user must be aware of the parallelization process and needs to learn the added classes to write parallel programs. Some implementations make this process simpler by providing an interface that is similar to standard MPI such as JOPI. Another drawback is loss in efficiency due to the overhead introduced to support remote objects and message-passing. This overhead is usually higher for systems that use RMI such as Ajents and Babylon. In addition, using class libraries limits the features that can be provided and flexibility in development.
3. Providing seamless parallelization for multi-threaded applications. This approach does not require programmers to be aware of the parallelization process (or at most have minimum involvement). Any Java program that uses multi-threading can be run in parallel on a distributed environment. Examples of this approach are the cJVM from IBM, JavaParty and ProActive. Here the system is designed to provide the user with an easy way to use multiple processors without dealing with the details of the process. The main advantage is that existing multi-threaded applications can be run on these systems without any changes as in cJVM or with minimum changes as in JavaParty and ProActive. One disadvantage in this approach is that optimizations in communications and locality are difficult. The IBM cJVM project has the disadvantage of having a modified JVM, which means that it does not easily support portability and heterogeneity. Moreover, the other two projects do not change the JVM, but require more user involvement such as defining the remote objects in JavaParty for specifying the parallel parts and the machines that should be used in ProActive.

From another viewpoint, we can also classify the projects based on the method used for parallelization. Some projects provide shared address space (or shared object space) such as Titanium, JPVM, cJVM and JavaParty, and others provide some form of message-passing or object-passing interface such as ParaWeb, Ajents, Babylon and JOPI. Despite the approach taken and the implementation techniques used in these projects, the nature of a distributed environment imposes some limits on the performance of the parallel or distributed application. The major issue is the cost of communication since the processors are not closely coupled as in a MPP or SMP. Here the overhead makes such environments mostly suitable for coarse grain parallel applications where communication is minimized and the computation-to-communication ratio is high. This limitation should gradually be overcome by the advancements in the processing and communications technologies.

4.2 Open issues

This study shows the growing interest in having environments for high performance computing in Java. Many approaches and implementations were used, but there are many open issues to be addressed. The following is a discussion of some of the issues related to these projects.

1. Since all projects are based on a distributed infrastructure, they all experience some inevitable overhead. This overhead is mainly introduced by the distributed nature of the platforms used. Generally, some methods have to be used to migrate objects and exchange information. This was achieved by either using RMI or socket programming. A few projects such as cJVM and JavaParty tried to refine their techniques to reduce the overhead.
2. Benchmarking these projects is also a difficult issue since each one has a different approach and syntax for the parallelization process. Until now the available benchmarks are limited to specific implementations such as mpijava and JMPI which are written based on MPJ [8]. As for other projects, many have written their own benchmark applications, which makes the comparison among the results of different projects difficult and inaccurate. It will soon be necessary to have some general benchmarks that can be easily ported to measure and compare the performance of the different implementations of parallel Java.
3. Conforming (or not conforming) with MPI or MPJ is another debated issue. Not conforming with a standard allows the developers to freely exploit the object-oriented nature of Java to simplify the parallelization process, but at the same time creates a new set of API that the user needs to learn and it becomes difficult to benchmark. On the other hand, conforming to some standard like MPI will limit the capabilities of parallel Java, while providing a

familiar interface to the users and making benchmarking easier. Some projects tried to join the two approaches by providing an MPI-like interface in addition to object-passing methods.

4. Legacy applications written in other languages such as C and FORTRAN need to be considered. Do we want to port such applications to Java? Alternatively, do we need to link Java with these applications? Many approaches may be used to solve this problem. One example is the UIUC project, which tries to build a comprehensive environment that can link and execute parallel modules written in different languages. The issues of efficiency and scalability become important in such implementations.
5. The security of the participating machines must also be considered. To run parallel Java programs on multiple machines, we allow users to upload their programs and execute them on the remote machines. In this case, we must consider the possibilities of malicious programs. Almost all projects have not addressed this issue. The JOPI system provides a starting point to providing security for the participating machines by making user threads run on the remote machines from memory and restricting them from accessing all other resources on that machine. More measures need to be considered to enhance security.
6. The performance of JVM was considered an important factor in deciding whether to use parallel Java. A few years back Java programs were many times slower than similar C programs. Currently with the major enhancements of JVM, this gap is closing very quickly and the performance of Java programs is becoming close to that of C programs. This has reduced the emphasis on this issue, but there is still more room for enhancements and better performance of Java.
7. Scheduling, dynamic load balancing and fault tolerance need to be addressed. Many parallel Java implementations do not consider these issues or only slightly touch on them. Since parallel Java is targeted for heterogeneous systems, were reliability is relatively low and performance of participating machines vary, these issues must be considered in more details and efficient algorithms and protocols must be designed to achieve them.

These are some of the issues to be addressed for a parallel Java implementation to be successful. The more features an implementation can offer the better chances for it to be successful and widely used.

Product Name	**Main Features**	**Approach used**	**User involvement**	**JVM Compatibility**
Titanium, Univ. California-Berkley	Java dialect. Scientific computing.	Language (compiles to C)	Must learn new language	Not compatible
ParaWeb, Waterloo & York	Runs parallel java programs on heterogeneous systems	Class library / run time machine modifications	Need to learn class methods	Java interpreter is modified
Ajents, Waterloo & York	Provide object migration. Uses RMI	Class library	Need to learn class methods	Compatible
Babylon, Waterloo & York	Adds scheduling and load balancing features	Class library	Need to learn class methods	Compatible
JPVM, Univ. of Virginia	Provides native parallel environment	Create new Java virtual machine	Need to know PVM	Not compatible
UIUC Project	Multi-language parallel programs support. Remote objects and load balancing	Combine different languages. Uses Converse and JNI	Need to know how to use the system libraries	Not compatible
ProActive, Université de Nice - Sophia Antipolis	Active objects. Migration. Based on RMI	Class library. Creates remote thread for objects.	Need to define active objects	Compatible, no preprocessing needed
JavaParty, University of Karlsruhe	Distributed applications Uses RMI	Transparent parallelization of multi-threaded applications	Need to write multi-threaded programs	Compatible Pre-compiler needed
cJVM / IBM	Creates single system image to distribute multi-threaded applications. Modified JVM	Transparent parallelization of multi-threaded apps.	Need to write multi-threaded programs	Not compatible
UNL – Project	Uses Software agents. JOPI	Class library	Need to learn JOPI (similar to MPI)	Compatible

Table 2. Summary of projects studied.

5. Conclusion

This survey provides a brief study of some research projects that are interested in providing parallel and distributed environments for java. Most projects target heterogeneous systems and clusters because Java is machine independent. The projects selected are representative of the different approaches taken in this area. Each of them has its own unique features, advantages and disadvantages, but they all aim towards the goal of having a parallel and distributed Java. We observed that almost all projects follow one of three approaches: Utilizing the available infrastructure or building a different JVM, extending Java with class libraries for parallel programming, and providing seamless transparent parallelization of multi-threaded applications. The research in this area still needs to address a number of open issues to be able to provide a robust, reliable and scalable parallel Java environment.

Acknowledgements

This study was partially supported by a National Science Foundation grant (EPS-0091900) and a Nebraska University Foundation grant, for which we are grateful. We would also like to thank other members of the secure distributed information (SDI) group and the research computing facility (RCF) at the University of Nebraska-Lincoln for their continuous help and support.

References

[1] Al-Jaroodi, J., Mohamed, N., Jiang, H. and Swanson D., "Agent-Based Parallel Computing in Java - Proof of Concept", Technical Report TR-UNL-CSE-2001-1004, http://www.cse.unl.edu/~nmohamed/jAgent/

[2] Aridor, Y., Factor, M., & Teperman A., "cJVM: a Single System Image of a JVM on a Cluster", IEEE International Conference on Parallel Processing, 1999.

[3] Aridor, Y., Factor, M., & Teperman A., "Implementing Java on Clusters", Technical Report, IBM Research Lab in Haifa, MATAM, Advanced Technology Center, Haifa 31905, ISRAEL.

[4] Aridor, Y., Factor, M., Teperman A., Eilam, T., & Schuster, A., "Transparently Obtaining Scalability for Java Applications on a Cluster", Journal of Parallel and Distributed Computing special issue – Java on Clusters, June 2000.

[5] Aridor, Y., Factor, M., Teperman A., Eilam, T., & Schuster, A., "A High Performance Cluster JVM Presenting a Pure Single System Image", ACM java Grande conference June 2000.

[6] Baker, M, Carpenter, B., Fox, G., Hoon Ko, S., and Lim, S., "mpiJava: An Object-Oriented Java interface to MPI," School of Computer Science, University of Portsmouth and Syracuse University, January, 1999 www.npac.syr.edu/projects/pcrc/papers/ipps99/paper/paper.html & www.npac.syr.edu/projects/pcrc/mpijava/mpijava.html

[7] Brecht, T., Sandhu, H., Shan, M. and Talbot, J., "ParaWeb: Towards World-Wide Supercomputing," Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland, pp. 181-188. Sep., 1996 http://bbcr.uwaterloo.ca/~brecht/papers/html/paraweb/welcome.html

[8] Bull, J. M., Smith, A., Westhead, M. D., Henly, D. S. and Dary, R. A., "A Benchmark Suite for High Performance Java" ,in Concurrency – Practice and Experiencre, 12, p375-388, 2000

[9]Carpenter B., et al., "MPI for Java: Position Document and Draft API Specificaton", Technical report JGF-TR-03, http://citeseer.nj.nec.com/cache/papers/cs/12456/http:zSzzSzwww.npac.syr.eduzSzprojectszSzpcrczSzreportszSzMPIpositionzSzposition.pdf/carpenter98mpi.pdf

[10] Crawford III, G., Dandass, Y. and Skjellum, A., "The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users," Web Technology Group, MPI Software Technology, Inc. (MSTI), http://www.mpi-softtech.com/publications/JMPI_121797.html

[11] Ferrari, A. J., "JPVM: Network Parallel Computing in Java," Technical Report CS-97-29, Dept. of Computer Science, Univ. of Virginia, December, 1997 http://www.cs.virginia.edu/~ajf2j/jpvm.html

[12] Izatt, M., "Babylon: A Java-based Distributed Object Environment, M.Sc. Thesis, Department of Computer Science, York University, July 2000.

[13] Izatt, M., Brecht, T. and Chan, P., "Ajents: Towards an Environment for Parallel Distributed and Mobile Java Applications," in ACM 1999 Java Grande Conference, Jun. '99

[14] Laxmikant, V. and Krishman, S., "Charm++: A Portable Concurrent Object Oriented Systems Based on C++," Proceedings of OOPSLA '93 Conference on Object Oriented Programming, Systems, Languages and Applications."

[15] Philippsen, M. and Zenger, M., "'JavaParty: Transparent Remote Objects in Java,", in Concurrency: Practice & Experience, Volume 9, Number 11, pp 1225-1242, Nov. 1997.

[16] Squyres, J., Willock, J., McCandless, B., and Rijks, P., "Object Oriented MPI (OOMPI): A C++ Class Library for MPI," in '96 POOMA Conference.

[17] ABC++ (1996): www.kfa-juelich.de/sam/cxx/parallel/abcplusplus.html

[18] Design and Implementation of Parallel Java with Global Object Space, at University of Illinois at Urbana Champaign (1997), web (Aug. 2001): http://charm.cs.uiuc.edu/papers/ParJavaPDPTA97.html

[19] IBM project cJVM (Aug. 2001): http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html

[20] JavaParty (Sep. 2001): http://wwwipd.ira.uka.de/JavaParty/

[21] OOMPI (Aug 2001): www.mpi.nd.edu/research/oompi

[22] ProActive (Aug. 2001): www-sop.inria.fr/sloop/javall/

[23] Official RMI web pages at Sun Microsystems (Aug. 2001): http://java.sun.com/products/jdk/1.1/docs/guide/rmi/index.html

[24]Serialization (Aug. 2001): http://java.sun.com/j2se/1.3.0/docs/api/java/io/Serializable.html

[25] Titanium (Jan 2001): http://www.cs.berkeley.edu/Research/Projects/titanium/