AN OVERVIEW OF PARALLEL AND DISTRIBUTED JAVA FOR HETEROGENEOUS SYSTEMS: APPROACHES AND OPEN ISSUES

JAMEELA AL-JAROODI, NADER MOHAMED, HONG JIANG, AND DAVID SWANSON *

Abstract. Java is gaining considerable recognition as the most suitable language for developing distributed applications in heterogeneous systems due to its portability and machine independence. However, standard Java does not provide easy-to-use features for parallel application development. Therefore, considerable research has been conducted and is underway to provide users with tools and programming models to write parallel applications in Java. This paper reviews a number of representative research projects and outlines the primary approaches used in these projects that enable Java to provide high performance parallel and distributed computing in heterogeneous systems. The study shows that most projects fit within one of the following parallel programming models: (1) message (or object-) passing, (2) distributed shared address (or object), (3) multi-threaded, and (4) transparent (or towards seamless) parallelization. Within these categories, the different implementation approaches are discussed. The paper also identifies and discusses a number of related problems and open issues such as benchmarks, porting legacy applications, distributed environment overhead and security.

Key words. parallel Java, programming languages, heterogeneous systems

AMS subject classifications. 68N19, 68N15

1. Introduction. Clusters, computational grids and heterogeneous networked systems can provide processing power comparable to special-purpose multi-processor systems for a fraction of the cost. Moreover, it is essential to have application software that can support such systems and provide the user with transparent and efficient utilization of the multiple resources available. Java emerges as a natural development environment for such architectures because it is portable, extendible and currently provides basic features that support distributed application development. However, standard Java is still not suitable for efficient parallel programming.

This paper studies and classifies a number of representative research projects that empower Java with parallel and distributed capabilities for clusters and heterogeneous networked systems. The classification is based on the programming model used. Within each model, projects are compared in terms of the implementation approaches, the level of user involvement, and the compatibility with the Java virtual machine (JVM). In addition, the paper discusses some of the problems, open issues and challenges facing such projects.

The paper provides some background information in section 2. Section 3 reviews the projects and classifies them in categories based on the programming models they embody. A discussion of the primary approaches used in these projects is presented in Section 4, which also identifies the problems and open issues in the area, while Section 5 concludes the study.

2. Background. Java in its current state provides features and classes that facilitate distributed application development. However, the development process of large scale distributed applications is usually very complex and time consuming. Some of the features Java provides are:

1. The reflection API, which represents, or reflects, the classes, interfaces, and objects in the current Java Virtual Machine [15].

*Department of Computer Science and engineering, University of Nebraska-Lincoln, 115 Ferguson Hall, Lincoln, NE 68588-0115, ([jaljaroo, nmohamed, jiang, dswanson]@cse.unl.edu)
2. Object serialization [45], which is used to store and retrieve objects in a serialized form by representing the state of objects using byte streams in sufficient details that allows for reconstructing the object(s) [15].

3. The Java class loader is responsible for loading the Java classes (bytecode) onto a JVM. Java allows programmers to override the default class loader by writing their own method for class loading. This is an important feature in Java for facilitating remote and dynamic class loading in a distributed environment.

4. Sockets, in Java, provide programmers with the flexibility to write efficient distributed applications, but they tend to make the development process lengthy and complex due to the low level details that need to be attended to.

5. Java Native Interface (JNI) [27] is a standard programming interface for writing Java native methods and embedding the JVM into native applications, thus making the application more efficient on the target machine. This provides binary compatibility of native method libraries across all JVM implementations on a given platform. However, using JNI compromises the portability of the Java application since parts of the code will be machine dependant.

6. Remote Method Invocation (RMI) [43] was introduced as a more user-friendly 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. However, this layer increases the costs of communications.

These same features can be used to develop parallel applications in Java. However, the process becomes even more complex and requires considerable effort to handle not just the communication aspects, but also the synchronization and process distribution, to mention just a few. In addition, some of these features are inefficient or introduce high overhead that offsets the efficiency of the parallel application. Therefore, some research groups have tried to enhance or modify them for their projects.

On the other hand, message-passing provides other programming languages such as C with a simpler tool to develop parallel applications in a distributed environment. The most well known standard for message-passing is the Message Passing Interface (MPI) [33]. MPI provides a number of library functions to exchange messages among processes, such as point-to-point and group communication primitives, synchronization and other functions. MPI-2 is an extension of MPI-1, adding more functionality such as process creation and management, extended collective operations, I/O, and additional language bindings such as C++ bindings.

Object-Oriented MPI was introduced recently to provide C++ programmers with abstract message-passing methods. A number of extensions were developed to provide object orientation for C++ and FORTRAN 90, such as OOMPI [46, 37], Charm++ [31] and ABC++ [8]. More recently, with the success of Java as a programming language for regular and distributed applications, some effort has been made to provide extensions of MPI that can be used in Java. The Java Grande Forum [26] has developed a draft standard for message-passing in Java (MPJ) [17] based on MPI.

3. Programming Models. In this section, we discuss the different programming models for parallel and distributed application development. Figure 3.1 shows a logical view of these models in increasing level of user involvement and awareness of the parallelization process, and the efficiency of the system adopting to the model. The message-/object-passing model is the most efficient in terms of computation performance, but requires full user awareness of the parallelization details (e.g. explicit
data distribution). On the other hand; while transparent parallelization tries to completely hide the parallelization details from the user, it is arguably the least efficient in terms of resource utilization and speedup. The models’ logical view also shows the implementation dependencies among the models. In a distributed environment, message-/object-passing is essential to support the other models. In addition, it is possible to implement each model by utilizing the features and functionality of the model(s) below. This may explain the great interest in developing and optimizing message-/object-passing models for Java in order to benefit future implementations of the higher-level models. The sub-sections that follow will discuss the different research projects in light of these models. As mentioned earlier, the projects discussed here are a representative subset of the available projects and is by no means a comprehensive list.

![Diagram](image)

Fig. 3.1. An overview of the organization of the programming models used for parallel Java (the arrow indicates increased user awareness (involvement) and increased efficiency)

3.1. Information Passing. In this category, systems provide certain mechanisms for some form of information exchange between processes, as in message-(object-) passing. This approach requires the utilization of a run-time support system to handle the application deployment and process allocation, in addition to the message or object exchanges between the participating machines. This environment can be implemented in many different ways such as using a pure Java implementation based on socket programming, native marshaling, RMI, utilizing the Java native interface (JNI), Java-to-C interface (JCI), parallel virtual machine (PVM) and others. In terms of the API provided, a number of systems were found to have tried to comply with MPI and MPJ [17], while others were based on a new set of class libraries for message (object) passing interfaces.

3.1.1. Java Object-Passing Interface [1, 32]. Developed at The University of Nebraska-Lincoln, the Java Object Passing Interface (JOPI) [32] provides the user with a class library of APIs very similar to the MPI and MPJ interface. Moreover, JOPI also exploits the object-oriented nature of Java by exchanging objects instead of data elements, which simplifies the process of writing parallel programs and facilitates the exchange of complex structures and logic. JOPI is a pure Java implementation and
the applications written with JOPI can execute on any JVM. Furthermore, the inter-
process communication is implemented using socket programming to ensure efficiency
and flexibility for the parallel application.

A run-time environment to support the parallel programming capabilities in JOPI
is provided [1]. Using this environment, parallel Java applications written with JOPI
can execute on homogeneous multi-processor systems or on heterogeneous systems.
The system is portable, which makes it possible to utilize different machines of varying
architectures to execute the user applications. Software agents [1] were used to coor-
dinate and manage the parallel processes and to schedule multiple user jobs among
the available processors. The agents help deploy and run the user processes on the re-
move machines as threads from the memory. This approach reduces the I/O overhead,
consumes fewer resources, and enhances the system security.

3.1.2. University of Waterloo and York University Research Projects.
A series of projects were developed here that facilitate parallel Java application de-
velopment:
ParaWeb [13] allows users to utilize the Internet computing resources seamlessly. It
enables users to upload and execute programs on multiple machines on a heteroge-
neous system. Using ParaWeb, clients can download and execute a single Java appli-
cation 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 execu-
tion of threads and provides communication using message-passing.
2. Java Parallel Runtime System (JPRS): the Java interpreter is modified to
provide the illusion of a global shared address space for the multi-threaded
application.

Ajents [24] 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 and uses Java security features
to protect the servers. Ajents provides many features such as remote object creation,
remote class loading, asynchronous RMI, object migration, and checkpointing, roll-
back and restart of objects.

Babylon [23], a Java based system to support object distribution, inherits Ajents’
features, in addition to a few new features. It allows object creation and migration at
any time, seamlessly handles arrival and departure of compute servers, and provides
I/O through the originating machine.

3.1.3. MPIJ - MPI for Java [34]. MPIJ was built as part of the DOGMA
project [19], but it can be used as stand alone system. This is a pure Java implemen-
tation of message-passing interface for Java and it is compliant with MPJ. The MPIJ
communication is built using native marshaling, which provides efficient communica-
tion primitives. The pure Java implementation makes MPIJ portable. Another useful
feature of MPIJ is that it is independent of the application frameworks; therefore, it
can be utilized to support different distributed applications such as DOGMA.

3.1.4. CCJ - Collective Communication in Java [36]. CCJ adds classes to
Java to support MPI-like message-passing and collective communication operations.
CCJ utilizes the object-oriented framework in Java to provide these operations. CCJ is
a pure Java implementation on top of Manta RMI, which is a modified implementation
of RMI on Myrinet. The use of Manta RMI reduces the overhead and utilizes the
faster Myrinet infrastructure.
3.1.5. **JPVM - Java Parallel Virtual Machine** [21]. Java Parallel Virtual Machine is a PVM-like library of object classes implemented purely in Java to achieve portability. The main goal is to enable a system to utilize the available computing resources in a heterogeneous system. It allows explicit message-passing 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 task creation and communication overhead is high, which implies that JPVM is most suitable for coarse grain parallelization.

3.1.6. **HPJava Language** [39, 22]. HPJava is being developed at the Syracuse University under the Parallel Compiler Runtime Consortium (PCRC) [39]. HPJava [22] is a dialect of Java for message-passing parallel programming, specifically designed for SPMD programming with distributed arrays added as language primitives. By design, applications written in HPJava can be preprocessed straightforwardly to standard Java with calls to kernel runtime functions. Java bindings of various runtime functions have been implemented and one of the usable components of the HPJava environment is the mpiJava [9, 35] binding of MPI. mpiJava uses JNI to link the parallel Java constructs and methods to the MPI library.

3.2. **Shared Address Space.** Here we discuss the systems that provide parallel Java capabilities through the shared address space model or the shared object model. In both cases, the parallel application is given the illusion of having a single address or object space where all data or objects are available to all the participating processes. Using a distributed shared address or object space, the user has less concern with the particular details of communicating information. However, it is still necessary to provide some parallelization information and directives in the application. The underlying infrastructure can be implemented in different ways, for example, using an existing distributed shared memory (DSM) system, or utilizing a message- or object-passing infrastructure. The systems discussed here used different approaches to handle the various issues of shared space such as information (data or objects) integrity and consistency, synchronization and coherence.

3.2.1. **Titanium** [47]. Developed at the University of California-Berkeley, Titanium is a Java dialect used for large scale scientific computing, where applications run in a shared address space. It provides parallelization primitives in a Java-like language, including immutable classes, flexible and efficient multi-dimensional arrays, and distributed data structures. One advantage is that programs written for shared memory can be executed in a distributed system without modification. The Titanium compiler compiles Titanium programs into C, thus it is not compatible with JVM; however, it inherits some of the safety features of Java.

3.2.2. **UIUC project** [29]. A research group at the University of Illinois at Urbana-Champaign has been working on a prototype extension of Java to provide dynamic creation of remote objects with load balancing, and object groups [29]. The language constructs, based on those of Charm++ [31], provide a shared address space. The parallel Java extension is implemented using the Converse interoperability framework [28], which makes it possible to integrate parallel libraries written in Java with modules in other parallel languages in a single application. Existing libraries written in C and MPI, Charm++, PVM, etc. can be utilized in a new application, with new modules written in Java using the provided parallelization runtime library.

The system is designed for multi-lingual parallel programming. To achieve parallelism, proxy objects and serialization are utilized in addition to asynchronous remote
method invocation and JNI to interface with converse messaging layer. The main implementation goals of this system are to minimize native code and copying.

3.2.3. PARASLAX [38]. Paraslax is a collection of Java packages that provide distributed shared object environment. The interface allows users to define and share objects on remote nodes and provides efficient consistency protocols. The code for a shared object is similar to an ordinary object with some modifications using Paraslax classes and methods.

3.3. Multi-Threading. Many research projects aim to provide seamless utilization of a distributed environment by executing multi-threaded programs on multiple connected machines. The main goal here is to be able to run concurrent multi-threaded applications in parallel without having to modify or rewrite them. This requires the system to transparently distribute the threads among the different processors without any user involvement. This is made possible by the inherent concurrency of multiple threads that can be translated into parallel processes on the distributed environment. In this case, the implementation issues are similar to the shared space model in the sense that all data and objects used by the threads need to be sharable. The underlying run-time support requires data sharing or exchange mechanisms to provide thread distribution and information sharing.

3.3.1. cJVM - Clustered JVM [4, 5, 6, 7, 18]. cJVM is a clustered Java virtual machine that allows multi-threaded applications to run on 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 of having remote and consistent replicated objects on different nodes.

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 employed. All these techniques enhance data locality by using caching based on locality of execution and object migration. In addition, to enhance data locality, the master copy of an object is placed where it will be used not where it was created. cJVM is a new JVM that replaces the standard JVM.

3.3.2. JavaParty [25, 40]. JavaParty provides facilities for transparent remote objects in Java and allows easy porting of multi-threaded Java programs to distributed systems such as clusters. The JavaParty environment can be viewed as a Java virtual machine that is distributed over several computers. 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. JavaParty extends the Java language with one modifier, called remote, to declare a JavaParty remote class or thread. The fields and methods of a remote object instantiaated from a remote class can be accessed transparently, while the JavaParty environment deals with locality and communication optimizations.

The JavaParty environment uses a pre-processor and a runtime system. The pre-processor translates the JavaParty source program to Java code and RMI hooks. The runtime system is a set of components distributed on all the nodes with a central component, called RuntimeManager, which maintains the locations of contributing
node objects. To reduce the access latency to a remote object while maintaining compatibility with the JVM, different optimization efforts were attempted, including more efficient object serialization and optimized RMI (KaRMI).

3.3.3. Hyperion [3]. Developed at The University of New Hampshire, Hyperion is an automatic distribution framework. It is aimed towards high performance execution of multithreaded Java applications on distributed systems. Hyperion consists of two parts: a Java bytecode-to-c compiler that compiles the Java classes into native C code, and a portable run-time system that is used to facilitate the communication and distribution of the generated code. Using Hyperion, a multithreaded Java application can be compiled and linked with the run-time system and then executed over a distributed shared memory system, thus alleviating the burden of explicitly parallelizing the application for the distributed environment. In addition, Hyperion provides a round-robin type of load distribution (of active threads) to achieve a basic level of load balancing. However, using the native code limits the portability of Hyperion to a set of predetermined UNIX systems and defies the original purpose for using Java.

3.4. Transparent (Seamless) Parallelization. In this category, some systems provide transparent parallelization of Java programs written in standard Java by modifying the JVM, while others utilize preprocessors to achieve this goal. Still others provide seamless utilization of resources or communication mechanisms to simplify the parallelization process. In general, the systems introduced in this category aim to hide the parallelization process details as much as possible, in an effort to get closer to the fully transparent parallelization of sequential applications. Thus, they try to relieve the developer of all details of parallelizing the application and of running existing applications in parallel without (or with minor) modifications. Again, a run-time support is needed to execute the generated parallel programs. The run-time support may be built from scratch or utilizing some facilities provided in the infrastructures described in the above three categories. For example, a distributed shared memory (DSM) system can be used to support the execution of a preprocessed parallel code.

3.4.1. ProActive [10, 16, 41]. ProActive includes a library for parallel, distributed, and concurrent (PDC) programming in Java. It provides a metacomputing framework to convert an object to a thread running in a pre-defined remote address space. Objects are classified into passive objects (non-thread objects) and active object (thread objects). 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 and the results will be transparently returned to the caller address space.

A sequential Java program can be transformed to a distributed program by converting some of the passive objects to active objects using the ProActive APIs. The rest of the sequential code requires no changes. Asynchronous RMI is used to allow the main thread to continue its execution without waiting for the result. The invocation of active object method immediately returns a future object, which is a reference to where the result of the method invocation will be placed. The caller thread will be suspended when it needs to use the result of the previously invoked remote method. This is called wait-by-necessity, which is a data-driven synchronization mechanism among the distributed threads. In addition, ProActive provides active object migration and group communication. Moreover, the latest releases of ProActive provide a framework using XML and monitors for supporting dynamic code loading.
on dynamically changing environments such as the Grid systems.

3.4.2. **JAVAR [12] and JAVAB [11]**. Developed at the Syracuse University, JAVAR [12] is a prototype restructuring preprocessor that parallelizes existing Java code by converting loops and recursive calls into multi-threaded structures to run in parallel. In the same spirit, JAVAB [11] is a prototype preprocessor that parallelizes loops in the Java bytecode. Similar to JAVAR, JAVAB generates a multi-threaded version that can then be executed in parallel.

3.5. **Others Approaches**. Some of the systems we encountered could not be classified directly under any of the four main categories identified above. This is mainly due to the hybrid approaches they have taken. One example is the Do! System [30], which transforms multithreaded applications into distributed applications while requiring user involvement in the process. In Do!, the user needs to use the classes provided to identify the parallelizable threads and remote object mappings. This approach, however, hides the details of the distribution and communication from the user. Another example is the JavaSymphony [20], which provides flexible controls over hardware/software resources and load balancing. Although JavaSymphony provides explicit parallel programming APIs, as in the message passing model, it does not follow that model. Instead, it provides an independent set of APIs for parallel and distributed programming. Java Symphony provides a Java class library written entirely in Java, thus maintaining compatibility with the JVM. This library provides many features, such as access to system parameters, user controlled mapping of objects, asynchronous RMI and selective remote class loading.

4. **Classifications and Open Issues**. Although Java is very suitable for distributed and multi-threaded applications, the features available in Java for distribution are not fine-tuned for tightly coupled processes as in the conventional parallel programming. Lately, many research groups have started working on providing parallel environments for Java. Most of them, as described above, have targeted clusters and heterogeneous networks of workstations because of Java’s portability and machine independence. The projects, compiled in the table in Figure 4.1, are some typical examples of the different approaches and programming models identified in this area of research.

4.1. **Comparison and Classification**. Based on the programming models used, the available parallel Java systems are classified into the following four different groups:

1. Systems supporting message-passing or object-passing among parallel and distributed processes. In this group, each system provides its own interface for users to utilize message-passing or object-passing. Many choose to provide MPI binding for Java such that the interface becomes compatible with MPI and MPJ. However, this limits the utilization of the object-oriented nature of Java. One system, an agent-based parallel Java provides an interface for object-passing (JOPI). Others choose to use existing infrastructure and features, such as using JNI (mpijava), or linking to C MPI or other libraries (UIUC project). Still another provides a pure Java implementation to maintain portability. This approach requires using different techniques such as RMI (in CCJ), native marshaling (MPIJ), or sockets (JOPI) for communication. A pure Java implementation gives the system the advantage of portability since it will be possible to simultaneously execute the parallel program on a heterogeneous collection of systems.
##Overview of Parallel and Distributed Java Systems

2. Systems providing a shared address space or shared object space. These systems provide the user with mechanisms to write parallel Java programs that logically share some data or objects. Most of these systems required changes in the JVM, resulting in such systems becoming dependent on the modified JVM. A very small number of implementations, such as Paralax, attempted to keep the system compatible with the standard JVM by adding classes to handle the required mechanisms for making data or objects available on all remote machines.

3. Systems executing regular multi-threaded Java applications on multiple processors. In this case, the system transparently executes a multi-threaded

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Main Features</th>
<th>Approach used</th>
<th>User involvement</th>
<th>JVM Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Message-Passing and Object-Passing</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOPI, U. Nebraska-Lincoln</td>
<td>Uses Software agents. JOPI</td>
<td>Class library</td>
<td>Need to learn JOPI (similar to MPI)</td>
<td>Compatible</td>
</tr>
<tr>
<td>ParaWeb, Waterloo &amp; York</td>
<td>Runs parallel Java programs on heterogeneous systems</td>
<td>Class library / run-time machine modifications</td>
<td>Need to learn class methods</td>
<td>Java interpreter is modified</td>
</tr>
<tr>
<td>Agents, Waterloo &amp; York</td>
<td>Provide object migration. Uses RMI</td>
<td>Class library</td>
<td>Need to learn class methods</td>
<td>Compatible</td>
</tr>
<tr>
<td>Babylon, Waterloo &amp; York</td>
<td>Adds scheduling and load balancing features</td>
<td>Class library</td>
<td>Need to learn class methods</td>
<td>Compatible</td>
</tr>
<tr>
<td>MPIJ, Brigham Young U.</td>
<td>Pure Java, MPJ compliant, uses native marshaling</td>
<td>Class library</td>
<td>API similar to MPI</td>
<td>Compatible</td>
</tr>
<tr>
<td>CCJ, Indiana U.</td>
<td>Pure Java, uses Manta RMI, MPJ compliant, optimized group communication</td>
<td>Class library</td>
<td>API similar to MPI</td>
<td>Compatible</td>
</tr>
<tr>
<td>JMPI (commercial)</td>
<td>object-oriented bindings to MPI</td>
<td>JNI bindings to MPI</td>
<td>API similar to MPI</td>
<td>Not Compatible</td>
</tr>
<tr>
<td>HPJava, PCRC Group</td>
<td>MPJ compliant</td>
<td>JNI bindings to MPI</td>
<td>API similar to MPI</td>
<td>Not Compatible</td>
</tr>
<tr>
<td>JPVM, Univ. of Virginia</td>
<td>Provides native parallel environment</td>
<td>Create new Java virtual machine</td>
<td>Need to know PVM</td>
<td>Not compatible</td>
</tr>
<tr>
<td><strong>Shared Address (Object) Space</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Titanium, Univ. California-Berkeley</td>
<td>Java dialect. Scientific computing.</td>
<td>Language (compiles to C)</td>
<td>Must learn new language</td>
<td>Not compatible</td>
</tr>
<tr>
<td>UIUC Project</td>
<td>Multi-language parallel programs support. Remote objects and load balancing</td>
<td>Combine different languages. Uses Converse and JNI</td>
<td>Need to know how to use the system libraries</td>
<td>Not compatible</td>
</tr>
<tr>
<td>Paralax (commercial)</td>
<td>Pure Java, provides consistency protocols.</td>
<td>Uses TCP sockets, fixed No. of nodes.</td>
<td>Need to learn some API primitives</td>
<td>Compatible</td>
</tr>
<tr>
<td><strong>Multi-Threaded Programming Model</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clustered JVM, IBM</td>
<td>Creates single system image to distribute multi-threaded applications. Modified JVM</td>
<td>Transparent parallelization of multi-threaded apps.</td>
<td>Need to write multi-threaded programs</td>
<td>Not compatible</td>
</tr>
<tr>
<td>JavaParty, U. Karlsruhe</td>
<td>Distributed applications</td>
<td>Transparent parallelization of multi-threaded applications</td>
<td>Need to write multi-threaded programs</td>
<td>Pre-compiler needed</td>
</tr>
<tr>
<td>Hyperion U. New Hampshire</td>
<td>Distributes multithreaded applications on a DSM system</td>
<td>Transparent parallelization of multi-threaded applications</td>
<td>Need to write multi-threaded programs</td>
<td>Not compatible, compiles to C, runs on UNIX systems</td>
</tr>
<tr>
<td><strong>Transparent (Automatic) Parallelization</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProActive, Université de Nice - Sophia Antipolis</td>
<td>Active objects. Migration. Based on RMI</td>
<td>Class library. Creates remote thread for objects.</td>
<td>Need to define active objects</td>
<td>Compatible, no preprocessing needed</td>
</tr>
<tr>
<td>JAVAR.U. Syracuse</td>
<td>Parallelize loops and recursive calls in Java code</td>
<td>Preprocessor</td>
<td>Preprocess code before compilation</td>
<td>Depends on run-time system</td>
</tr>
<tr>
<td>JAVAR.U. Syracuse</td>
<td>Parallelize loops in bytecode</td>
<td>Preprocessor</td>
<td>Preprocess bytecode before execution</td>
<td>Depends on run-time system</td>
</tr>
</tbody>
</table>

**Fig. 4.1. Summary of the systems studied.**
program in parallel by distributing the threads among the participating processors. Some systems, such as cJVM, provide a different JVM that creates a single system image (SSI), thus hiding the details of the underlying infrastructure from the application. The main advantage of this model is that existing multi-threaded applications can run seamlessly in parallel without any (or with minor) modifications. However, a disadvantage in this approach is that optimizations for communication and locality are difficult. In addition, cJVM has the disadvantage of having a modified JVM, rendering its support for portability and heterogeneity difficult. While JavaParty does not change the JVM, it requires more user involvement (such as defining the remote objects).

4. Systems capable of transparent and relatively seamless parallelization of existing Java applications. Although this may be the most attractive model (from the application development viewpoint), it is the one that is the least explored. A system in this category should provide some mechanisms to transparently parallelize an application. However, some of the systems in this category require some help from the programmer to make it possible (as in ProActive). Prototypes of preprocessors are also available that try to parallelize loops and recursive calls in a Java code or bytecode. The transparent parallelization model is very attractive, but until now, there is no simple way to achieve it. The complexity and diversity of the application is one main barrier and maintaining efficiency is another challenge.

On the other hand, the systems discussed above may be examined in a different four-category classification, from an implementation point of view, as follows [2]:

1. Developing a run-time environment based on existing technologies and infrastructures. This approach utilizes current techniques such as JNI bindings to MPI, JCI, distributed shared memory (DSM) systems, among others, to support the parallel Java environment. An advantage of this approach is that most of the underlying technologies have been optimized for efficiency and widely used and tested. However, these implementations limit the use of parallel Java to the systems and platforms that support these techniques and make the parallel Java programs non-portable. Examples in this category include ParaWeb (the JPRS implementation), the project at UIUC, mpijava, and Hyperion.

2. Replacing the JVM by a modified version to support parallel and distributed Java. The advantage of this approach is the total control the developers have over the environment (The new JVM), thus enabling an efficient implementation. However, one major disadvantage is that the modified JVM will not be compatible with other JVMs, leading to loss of portability. In addition, enhancements or changes in the standard JVM cannot be easily incorporated into the new system. Moreover, adding more machines to the system becomes non-trivial. Some examples of this category are the JPVM that creates a new JVM for parallel processing and the Clustered JVM (cJVM).

3. Providing new parallel languages that are dialects of Java. The main advantage here is the ability to provide different functionalities in the new language without having to fully comply with the Java language specifications while keeping the desirable features in Java. The main disadvantage, again, is the machine dependence of the new language, hence making it difficult to port the applications to other platforms. Examples in this category are Titanium
4. Providing a pure Java implementation by extending Java with class libraries to provide explicit parallelization functions. Such implementations require some form of run-time support to exist on the participating machines. This approach preserves the portability and machine independence of Java, which enables the parallel application to run on different architectures, thus providing support for heterogeneity. Another advantage is that the addition of more machines to the system is effortless. One disadvantage is that users must be aware of the parallelization process and need to learn the added classes. Some implementations make this process simpler by providing an interface that is similar to MPI such as JOPI, MPIJ, and CCJ. Another drawback is the loss in efficiency due to the overhead introduced to support remote objects and message-passing. This overhead is higher for systems using RMI, such as Ajents and Babylon. In addition, using class libraries limits the features that can be provided and the flexibility in development. Some examples in this category include ParaWeb (the JPCL implementation), Ajents, Babylon, JOPI, CCJ and Parsons.

In addition, some of the projects provide mechanisms for dynamic class loading as part of the system or the support environment such as in JOPI, JavaParty, ProActive and Java Symphony, while others do not discuss the process/class deployment mechanisms. Regardless of the approaches 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 tightly coupled as in a MPP or SMP. Here the overhead makes such environments mostly suitable for coarse grain parallel applications where communication is relatively small and infrequent and the computation-to-communication ratio is high. This limitation should gradually be overcome by advancements in the processing and communications technologies.

4.2. The Open Issues. This study shows a steadily growing interest in creating environments for high performance parallel and distributed computing in Java. While many design and implementation approaches have been used by various research projects and prototypes, numerous problems and open issues remain to be addressed. The following is a discussion of some of the issues related to these systems.

1. Since all systems are based on a distributed infrastructure, they all experience some inevitable overhead introduced by the distributed nature of the system. Generally, some methods have to be used to migrate objects and exchange information. At the present, RMI and socket programming have been the most widely used methods for information exchange. A few projects such as cJVM, CCJ and JavaParty have tried to refine their techniques to reduce the overhead. Nevertheless, reducing communication overhead remains a difficult challenge.

2. The lack of a general agreement on what is a suitable implementation approach has led to many different implementations and various types of APIs. To further complicate the situation, the rapid advancement in the supporting (underlying) technology has meant that some implementations thought to be inefficient before could become efficient now or in the near future and the trade-off between simplicity and performance in implementations could be shifting immensely. For example, using RMI was considered inefficient by some, yet an improved RMI for specific implementations (such as KaRMI
and Manta RMI) has made it much more efficient while keeping the flexibility of development associated with RMI. Another example is the use of JNI to bind with MPI, which had to be done manually before research suggested an automated model to generate the JNI bindings.

3. Benchmarking research projects, especially with macro benchmarks and live applications, is difficult since each one has a different design, implementation approach and API. Until now, the available benchmarks are limited to micro benchmarks of specific operations or to specific implementations such as mpijava and JMPI, which are written based on MPJ [14]. Many others have written their own benchmark applications, which make the comparison of the results of different projects difficult and inaccurate. Therefore, it is necessary to have some general benchmarks that can be easily ported to measure and compare the performance of the different parallel Java systems.

4. Conforming (or not conforming) with MPI or MPJ is another debated issue. Not conforming to a standard allows the developers to freely exploit the object-oriented nature of Java to simplify the parallelization process. However, this creates a new set of APIs that the user needs to learn, making it 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 user and making benchmarking easier. Some projects have tried to combine the opposing approaches by providing an MPI-like interface and object-passing methods.

5. 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? Porting legacy codes would require a considerable amount of effort, which can be further increased by the many different approaches and APIs taken to design and implement the parallel Java environment. On the other hand, the alternative would limit the portability of the parallel Java programs due to links to machine dependant codes. An example of the second approach is the UIUC project. The issues of efficiency, portability and scalability become more important in such implementations.

6. The security of the participating machines and user applications must also be considered. To run parallel Java programs on multiple machines, users are allowed to upload their programs and execute them on the remote machines, where caution must be given to the possibility of malicious programs. The JOPI system, for example, provides a starting point to providing security for the participating machines. More measures need to be considered to enhance security and to protect the machines and users.

7. Scheduling, dynamic load balancing and fault tolerance issues 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, where reliability is relatively low and performance of participating machines vary significantly, these issues must be considered in more details by designing efficient algorithms and protocols to attack such problems.

These are some issues to be addressed for a successful design and implementation of a parallel Java environment. While it may be difficult, if not impossible, to address all these issues at the same time, a particular implementation might judiciously choose to emphasize more on some specific issues than others, depending on the available
underlying technology and infrastructure.

5. Conclusion. This paper conducted a survey that provides a concise study and classification of research projects involved in providing parallel and distributed environments for Java. Most of the studied systems target heterogeneous systems and clusters because of Java’s portability and machine independence. The projects selected are representative of the different approaches and programming models known in this area. While each of them has its own unique features, advantages and disadvantages, they all aim towards the goal of having a parallel and distributed Java. We observed that almost all projects follow one of the following programming models: (1) Message- or object-passing, (2) shared address (or object) space, (3) multi-threading, and (4) transparent/seamless parallelization. From an implementation point of view, we were able to classify these projects based on the following four implementation approaches: (1) Utilizing the available infrastructure, (2) building a different JVM, (3) providing pure Java implementation by extending Java with class libraries, and (4) building new Java dialects for parallel programming. The study further identified a number of problems and open issues in this area that remain to be addressed in order to provide a robust, reliable and scalable high performance parallel and distributed Java environment for clusters and heterogeneous networked systems.

Acknowledgments. This work was partially supported by a National Science Foundation grant (EPS-0091900) and a Nebraska University Foundation grant. We would like to thank members of the secure distributed information (SDI) group [44] and the research computing facility (RCF) [42] at UNL their continuous support.

REFERENCES

OVERVIEW OF PARALLEL AND DISTRIBUTED JAVA SYSTEMS


