JavaGrande: Java for Parallel Computing

Bryan Carpenter and Sung-Hoon Ko
NPAC at Syracuse University
Syracuse, NY 13244
dbc@npac.syr.edu, shko@npac.syr.edu

January 11, 2000

Abstract

Contents

1 Introduction

The initial growth of interest in Java was driven largely by its role in bringing a new generation of interactive pages to the World Wide Web. Undoubtedly various features of the language—compactness, byte-code portability, security, and so on—make it particularly attractive as an implementation language for applets embedded in Web pages and servlets embedded in servers. But it was always clear that the ambitions of the Java development team go well beyond enhancing the functionality of HTML documents. Many of their concerns, such as portability, execution in a heterogeneous network environment, and efficiency, mirror development in High Performance Computing world over a number of years [16, 14, 17, 27].

With Java positioned to become a standard programming language on the Internet, and scientific parallel processing edging towards network-based computation, it is natural to ask how these two technologies will interact. How suitable is Java for scientific computing, and do lessons from research in parallel computing have implications for the development of Java?

Popular acclaim aside, there are some reasons to think that Java may be a good language for scientific and parallel programming.

- Java is a descendant of C++. C and C++ are used increasingly in scientific programming. In recent years numerous variations on the theme of C++ for parallel computing have appeared. See, for example [11, 13, 20, 41]

- Java omits various features of C and C++ that are considered "difficult"—notably, pointers. Poor compiler analysis has often been blamed on these features. The inference is that Java, like Fortran, may be a suitable source language for highly optimizing compilers.

- Java comes with builtin multithreading. On a shared memory platform independent threads may be scheduled on different processors by a suitable runtime. Even without physical shared memory, mulitithreading can be very convenient in explicit message-passing styles of parallel programming.

The outline of this lecture is as follows. Section 2 provides a general overview of the Java Grande Forum, its activities, and the process. Java Grande Forum is addressing many of the issues that arise in using Java for High Performance Computing and Grande Applications. Section 3 and 4.3 of this lecture outline various options for parallel programming in Java–possible ways to express parallelism, and ways to handle inter–process communication. Finally in Section 5, we look at related work of Java Grande.

2 Java Grande Forum

The attractiveness of Java for scientific computing is being encouraged by bodies like Java Grande [3]. The first meeting of the Java Grande Forum was held at

Java '98 on February 28 and March 1, 1998, in Palo Alto, California. The Java Grande forum is motivated by the notion that Java has the potential to be a better language than previous languages such as C and Fortran for developing large science and Grande applications, particularly on distributed and high-performance computers. A Grande Application is an application of large scale nature, potentially requiring any combination of computers, networks, I/O, and memory. For examples, financial modeling, aircraft simulation, climate and weather, satellite image processing and earthquake prediction.

The major goal of the Java Grande Forum is to develop community consensus and recommendations for either changes to Java or establishment of standards for Grande libraries and services.

The forum is open and operates with a mix of small working groups and public dissemination and request for comments on its recommendations. Currently, two working groups are exist: Numeric Working Group and Concurrency/Applications Working Group.

The goal of the Numeric Working Group is to assess the suitability of Java for numerical computing, and to work towards community consensus on actions which can be taken to improve the language and its environment for applications with significant requirements for floating-point computation. In its initial assessment of Java, the working group has focused on five critical areas where improvements to the Java language are needed:

- floating-point arithmetic
- complex arithmetic
- multidimensional arrays
- lightweight classes
- operator overloading

Lightweight classes and operator overloading provide key components to proposed improvements for complex arithmetic and multidimensional arrays. Information about the Numeric Working Group activities is maintained at the JavaNumeric web page at http://math.nist.gov/javanumerics.

The goal of the Concurrency/Applications Working Group is to assess the suitability of Java for parallel and distributed computing and so-called problem-solving environments(applications) and to work towards community consensus on actions which can be taken to improve the language and environment. The focus has been on the Java-specific frameworks designed to support concurrent and high-performance computing methodology. The working group is organized as follows:

- **critical JDK issues**
  highest priority issues, mostly related to performance of RMI and Object Serialization
- **benchmarking JVM performance**

- **seamless(desktop) access to high-performance computers(computing portals)**
- **other parallel and distributed computing issues**
  the availability of MPI-like message passing facilities for programming networks of computers

Information about the Java Grande Concurrency and Applications Working Group activities can be found at http://www.javagrande.org and following the direct link to Working Groups.

3 Java for Scientific Parallel Programming Issues

3.1 Approaches to Parallelism in Java

Java already supports concurrency through the thread mechanism and monitor synchronization built into the language. In this lecture we are interested in truly *parallel* computation, involving multiple CPUs. Such parallelism could be introduced into Java in a number of ways.

It could be achieved through automatic parallelization of sequential code, but it is unclear why this easier for Java than for other languages. Alternatively, the Java virtual machine for a shared memory multiprocessor can schedule the threads of a multi-threaded Java program on different processors. Some success with these approaches in reported in [5]. For computation on a network(or distributed memory computer) realistic options include language extensions or directives akin to HPF, or provision of libraries—class libraries—to support task parallelism or data parallelism.

A popular approach in C++ has been to defer language extensions and concentrate on class library support for parallel programming. The similarities between the two languages suggest this may be a fruitful avenue in Java too. The success of this analogy is by no means automatic, however. Features such as *templates* and user-defined *operator overloading* make the C++ language inherently more customizable than Java. In C++ library-defined types can be used on an identical footing to primitive types—*inline* methods mean they can be almost as efficient as primitive types. Less importantly, but conveniently, new control constructs can often be simulated in C or C++ through use of macros. On grounds, presumably, of simplicity and transparency many of these features have been omitted from Java.

3.2 Communication in Java

The standard Java API provides a simplified interface to Internet sockets and RMI, a Java version of Remote Procedure Call (RPC).

The socket interface hides much of the ugly detail involved in socket-programming at the traditional C/Unix level. The *java.net* interfaces provides less flexibility than using the system calls directly. On the other hand, Java's built-in support for threads adds some flexibility in scheduling communications that is missing from raw C. Traditionally socket programming has not been a popular paradigm in the parallel processing world, where more successful schemes include

- Message-passing through language-level support or higher-level library interfaces [27].
- Data parallelism, which we take to mean the style of programming in which parallelism is achieved through operations on distributed arrays, with synchronization typically limited to bulk synchronization occurring naturally through collective array operations.
- Communication through shared memory or shared objects, involving some more or less intricate mechanism for inter-process synchronization.

Communication through remote objects(i.e RMI) is undoubtedly a natural and important paradigm in Java, especially for access to remote services. However, RMI is too restrictive and its overhead for dealing with network problems is too verbose for the cluster setting. It suffers from a significant startup time and marshaling/unmarshaling of data is more costly than in the socket method.

4 Message Passing with Java

A distributed memory system consists of processors and their local memories connected by interconnection network. Since there is no global memory, it is necessary to move data from one local memory to another by means of message-passing. This is typically done through *send/receive* API calls which must be written into the application program. Message-passing remains one of the most effective and widely used communication paradigms in parallel computing. In this section we discuss three approaches to message-passing in Java, in the context of a "scientific" application. The first approach is to use the socket interface in the standard Java API. The second is to work through Java's RMI. Finally the third is to work through a Java interface to the message-passing standard, MPI [27].

4.1 Java sockets

The Unix socket model is most suitable for programming client-server applications. Typical scientific parallel programs do not fit directly into this model. Before a SPMD program can start two conditions must obtain: a pool of symmetric peer processes must have been created, and each peer must be able to address a message to any other. Bootstrapping this situation typically involves one host starting remote invocations of the program (for example by using *rsh* or a specialized daemon). All the peers will create listening sockets, and all the port numbers must be broadcast somehow, then socket connections will be made.

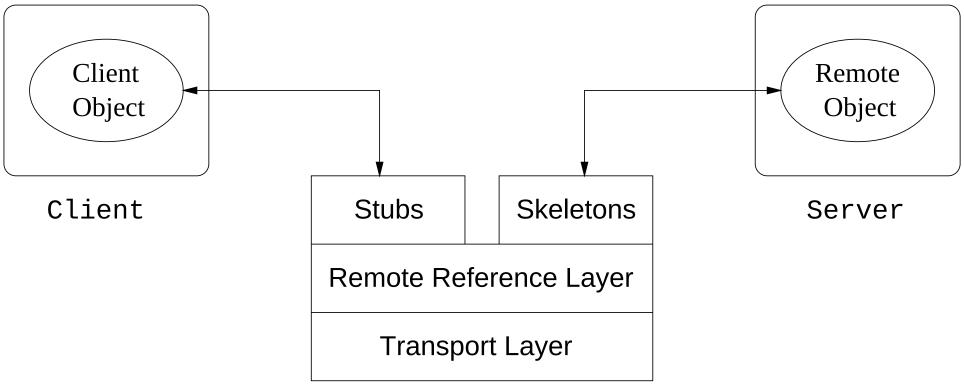

Figure 1: RMI Architecture.

Even with the simplified Java socket API, direct socket programming will remain unattractive to scientific parallel programming.

4.2 Java RMI

Java Remote Method Invocation(RMI) which is an object-oriented version of Remote Procedure Call(RPC), has been included with the Java Development Kit release 1.1. RMI provides an simple and powerful Java-to-Java communication model for invoking member functions on objects that exist in other Java virtual machines exactly as if it were local objects running in the same virtual machines. As most distributed processing is conceived, RMI is based on the Client-Server model. An overview of the Java RMI model is shown in figure 1.

The RMI architecture consists of three independent layers between the application program and the Java virtual machine. Each layer is independent and can be replaced by an alternate implementation. The three layers are the Stub and Skeleton Layer, the Remote Reference Layer, and the Transport Layer.

The respective applications reside on both the client and server machines. The client and server communicate with the RMI system through a stub and skeleton respectively. These methods reside on top of the Remote Reference Layer and Transport Layer, allowing the RMI system to handle the interface. When a single request occurs, it travels down through the layers on one computer and up through the layers at the other end.

The stub and skeleton layer is responsible for marshaling and unmarshaling the data and transmitting and receiving them to/from the Remote Reference Layer. The data are marshaled and unmarshaled using a parameter-passing mechanism called *object serialization*. Object Serialization is an essential feature needed by Java's RMI implementation. It provides a program the ability to read or write a whole object to and from a raw byte stream. Object Serialization allows Java objects and primitives and to be encoded into a byte stream suitable for streaming to some type of network or to a file-system. In order to provide this functionality, an object must implement the Serializable interface.

The Remote Reference Layer is responsible for carrying out the invocation. It can be behave differently depending on the parameters passed by the calling program. For example, this layer can determine whether the request is to call a single remote service or multiple remote programs as in a multi-cast.

The Transport Layer is responsible for setting up connections, managing request, monitoring them and listening for incoming calls.

High performance distributed computing can be done with Java RMI, but this has a few weakness since RMI is designed for client-server programming in Web based systems over slow networks. In parallel programming, fast RMI with low latency, high bandwidth, and better reliability are required. A better serialization is also needed for a fast RMI, since Java's object serialization often takes 25%-50% of the time [33] needed for a remote invocation.

4.3 Message Passing Libraries in Java

The most popular message-passing libraries are PVM and MPI. Several projects are under way to make message-passing packages to support programs written in Java. Current message-passing in Java can be classified into two categories. First category consists of Java based implementations that use Java as wrapper for existing message-passing libraries that are called through the Java native interface (JNI). The use of native methods achieves better performance and allows programmers to reuse existing codes. However, it restricts jobs to run only on system where both Java and corresponding message-passing library are supported.

The second category of Java based implementations use only pure Java libraries. Typically, this case the underlying communication libraries are Java's RMI or socket. The pure Java implementation obtains high degree of code portability among platforms but relatively lower performance. The performance problem is expected to be overcome with just-in-time (JIT) compilation and other compiler technologies.

In the next section we will discuss Java interface to two popular message-passing standard, PVM and MPI.

4.3.1 Parallel Virtual Machine (PVM) in Java

PVM developed at Oak Ridge National Laboratory is a message-passing system that permits a heterogeneous collection of networked computing systems to be used as a single large parallel machine. Thus large computational problems can be solved more cost effectively by using the aggregate power and memory of many computers. PVM consists of two parts: a daemon process, *pvmd*, that any user can install on a machine, and a user library that contains routines for initiating processes on other machine. PVM provides a rich set of dynamic resource management and process control functions (eg., load balancing). It also allows the development of fault-tolerant applications with host or task failure and has good interoperability between heterogeneous hosts. PVM supports programs written in C, C++, and Fortran. Currently there are two groups working on making PVM support programs written in Java.

- **Java-to-PVM interface**
  JavaPVM (or jPVM) [1] is an interface written using the Java native methods capability which allows Java applications to use the Parallel Virtual Machine (PVM) software. Since it is using native method, it limits cross-platform portability.

- **Pure Java implementation of PVM**
  JPVM [12] is a pure Java implementation of PVM. Therefore unlike JavaPVM, it is better matched Java programming styles and the standardization of the JVM makes the JPVM code much simpler to maintain across heterogeneous machine. The performance is poor compared to JavaPVM. The detail comparison of communication benchmark of native PVM, JavaPVM, and JPVM is reported in [42].

4.3.2 Message-Passing Interface(MPI) in Java

MPI [27, 21] is a library of functions and macros that can be used in C, FORTRAN, and C++ programs. MPI is intended for use in programs that exploit the existence of multiple processors by message-passing. MPI was developed by a group of researchers from academia, industry, and government. As such, it is one of the first standards for programming parallel processors, and it is the first that is based on message-passing. MPI provides a rich set of communication libraries, application topology and user defined data types. The first phase of MPI (MPI 1) was focused on developing a message-passing standard. The second phase of MPI effort (MPI 2) is underway adding parallel I/O, fault tolerance, language binding for C++ and additional communications. Currently there are several known efforts towards the design of early MPI interfaces for Java.

- **Automatic generation of Java wrappers to MPI**
  An approach comparable to our work to creating full Java MPI interfaces has been taken by University of Westminster [28, 18]. In their work Java wrappers were automatically generated from the C MPI header using Java-to-C interface generator (JCI). This eases the implementation work, but does not lead to a fully object-oriented API.

- **Pure Java implementations of MPI**
  MPIJ [2] is a pure Java based subset of MPI developed at Brigham Young University as part of the Distributed Object Group Meta-computing Architecture (DOGMA) [24] system. Since it is implemented in only Java, portability....

  jmpi [9, 10] is a pure Java implementation of MPI built on top of JPVM. Due to additional wrapper layer to JPVM routines, its performance is poor compared to JavaPVM.

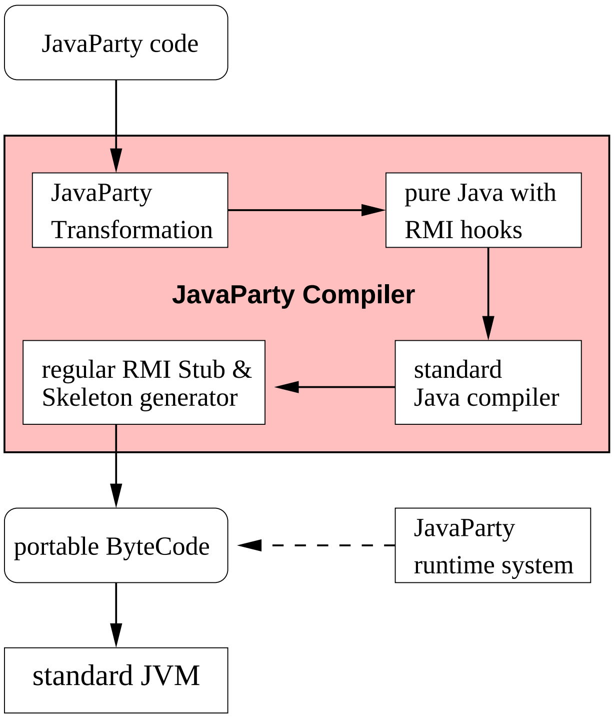

Figure 2: The JavaParty system.

JMPI [23] is a commercial effort underway at MPI Software Technology, Inc. to develop a message-passing framework and parallel support environment for Java. It targets to build a pure Java version of MPI-2 standard specialized for commercial applications.

5 Related Work of Java Grande

Currently several projects are under way that achieve increased bandwidth, reduced latency, and better reliability within workstation clusters using Java. In this section we briefly discuss some works related to Java Grande.

5.1 JavaParty

The JavaParty [34] system of the University of Karlsruhe, Germany, provides a mechanism for parallel cluster programming in Java. JavaParty adds remote objects to Java purely by declaration avoiding disadvantages of explicit socket communication and the programming overhead of RMI. The `remote` keyword is used to identify which objects can be called remotely. JavaParty is location transparent, i.e., the programmer does not need to map remote objects and

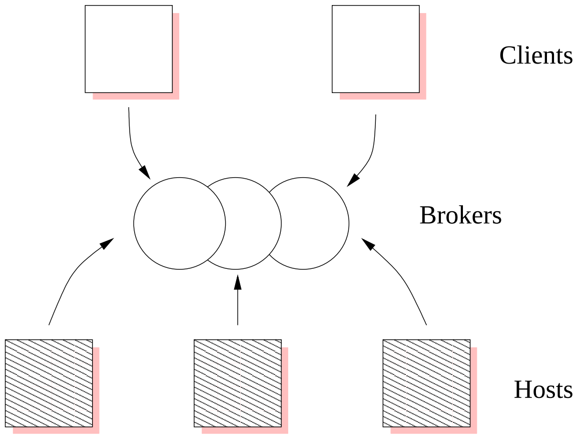

Figure 3: The Javelin Architecture.

remote threads to specific nodes of the network; compiler and runtime system deal with locality and communication optimization.

JavaParty is implemented as a pre-processing phase into Java compilers, EspressoGrinder [30] and Pizza [31]. As shown in the figure 2, JavaParty code is transformed into regular Java code plus RMI hooks. The resulting RMI portions are then fed into RMI compiler(stub & skeleton generator).

The runtime system consists of a central component, called *RuntimeManager*. In addition, each host runs a *LocalJP* that is registered at the manager. A host and its LocalJP can be added dynamically to the system. The manager knows all LocalJPs and knows the location of all class objects, i.e., for each class that is loaded the manager knows which host implements the static part. This information is replicated in the LocalJPs to reduce manager load. Neither manager nor LocalJPs need to know the location of individual remote objects. LocalJPs are needed to call constructors in class objects and to implement either side of migration.

JavaParty also supports object migration. For actually migrating an object, the runtime system offers two options. An object can move to the position of the caller or it can be moved to the location of different remote object. A migration is possible if no method is executing on the object, no other migration is in progress, and the object does not have its `resident` flag set. Then the object's internal state is serialized into a byte array which is sent to the runtime system on the target host. The receiving runtime system unpacks the byte array and returns an RMI stub reference to the runtime system of the original host. Upon receipt, the original host completes the migration.

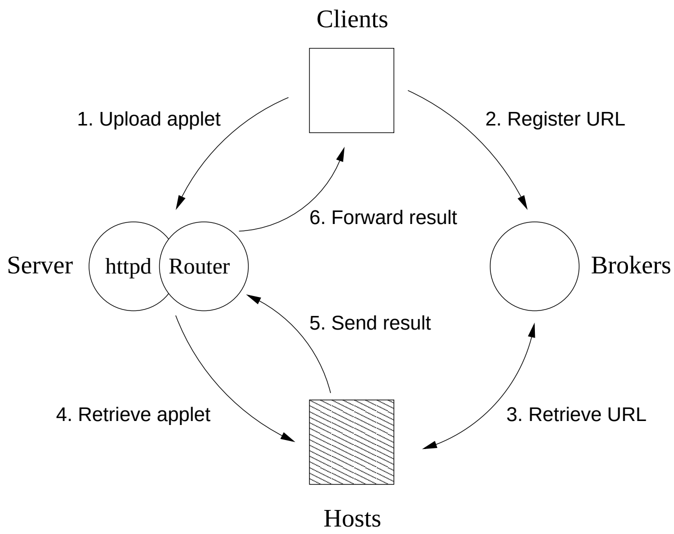

Figure 4: Steps involved in the remote execution of an applet.

5.2 Javelin

Javelin [8] of the University of California, Santa Barbara, is internet-based global computing infrastructures that support Java. In Javelin, there are three kinds of participating entities: clients, hosts and brokers. A *client* is a process seeking computing resources; a *host* is a process offering computing resources; a *broker* is a process that coordinates the supply and demand for computing resources. Figure 3 illustrates Javelin architecture. Clients register with the broker their tasks to be run; hosts register with the broker their intention to run tasks. The broker assigns tasks to hosts that, then, run the tasks and send results back to the clients.

One of the most important goals of Javelin is *simplicity*, i.e., to enable everyone connected to the Internet or an intranet to *easily* participate in Javelin. To this end, the design is based on widely used components: Web browsers and the portable Java. By simply pointing their browser to a known URL of a broker, users automatically make their resources available to host parts of parallel computations. This is achieved by downloading and executing an applet that spawns a small daemon thread that waits and "listens" for tasks from the broker. The simplicity of this approach makes it easy for a host to participate–all that is needed is a Java-capable web browser and the URL of the broker.

In Figure 4 shows the steps involved in the remote execution of an applet. These steps are:

1. The client uploads the applet and an embedding HTML page on an HTTP server. Clients running their own HTTP server may skip the first step.

2. The client registers the corresponding URL with a broker.

3. The host registers with the broker its intention to execute tasks, and retrieves the URL of a task.

4. The host downloads the HTML page from the HTTP server, and executes its embedded applet.

5. The host stores the result at the server site. If communication between cooperating hosts is required, messages are stored at, and retrieved from, the server site.

6. The client retrieves the result.

The server is an HTTP server that performs two separate functions. First, it stores clients' applets and serves them to hosts; secondly, it provides point-to-point communication among hosts by serving as smart router.

Tasks are represented as applets embedded in HTML pages. This design decision implies certain limitations due to Java applet security: E.g., all communication must be routed through the broker and every file access involves network communication. Therefore, in general, coarse-grained applications with a high computation to communication ratio are well suited to Javelin.

5.3 Efficient RMI and Object Serialization

One of the goals of Java Grande Forum is to investigate into methods to optimize Java RMI and Object Serialization performance. RMI implementations are built on top of TCP/IP, which was not designed for High Performance Computing. Several research groups are pursuing research to improve efficiency of Java RMI and object serialization implementation and its performance.

As part of the JavaParty [34] project at the University of Karlsruhe, Germany, a more efficient RMI and object serialization have been studied. Philippsen et al [32] describe detailed analysis of RMI and object serialization and it's performance problem. They also suggest modifications to improve RMI and object serialization. Based on these suggestions, they implement more efficient version of RMI and object serialization called KaRMI and UKA-serialization. KaRMI [29] improves the performance of RMI by clean interfaces between layers and the separation of RMI's design from transport level sockets. They also modify RMI to support non-TCP/IP communication networks. KaRMI cannot handle code that uses socket factory or port numbers. UKA-serialization [33] uses explicit marshaling/unmarshaling routines instead of reflection and a minimal encoding of type information to achieve better performance.

Henri Bal's group of Vrije University, Netherlands, study an alternative approach for implementing RMI, called Manta RMI [26], based on native compilation for better optimization. Manta [37] is compiler-based Java system and uses a native compiler that generates executable code rather than byte code. Manta system supports two different communication protocols. Between Manta nodes, it can communicate through Manta's own a fast RMI protocol using Manta's own serialization format. Manta node can communicate with another

JVM through a JDK-compliant protocol using Sun's serialization format. With Manta, the serialization code is generated by the compiler which make it possible to avoid the overhead of dynamic inspection of the object structure.

Krishnaswamy et al [25] improve RMI performance by adding object caching at the reference layer and using UDP(User Datagram Protocol)-based reliable message delivery protocol.

Another implementation of RMI is provided by Thiruvathukal et al. Reflective RMI(RRMI) [36] adopted the reflection feature of Java programming language to invoke remote methods. RRMI makes it possible of use RMI without stubs and skeletons. A remote object is created dynamically at the run time by implementing its own class loader on the server side. It also supports both synchronous and a asynchronous remote method invocations.

Raje et al implement ARMI (Asynchronous RMI) [35] which allows an asynchronous remote method to RMI. It achieve asynchronous communication by replacing the RMI client-side stub by adding thread mechanism.

NexusRMI [7] implements Java RMI on top of NexusJava [15], which is a Java interface to the Nexus communication system developed at Argone National Laboratory. NexusRMI also has support for object serialization of all object types except for exception objects.

At the group of the University of California, Berkeley, NinjaRMI [40] has been developed to support Ninja project [39]. NinjaRMI has some features that are not found in Sun's RMI including multiple communication protocols.

5.4 Other parallel and distributed computing in Java

Most Java based parallel systems can be classified into four classes.

- **Use Java as wrapper for existing frameworks.**
  Java/DSM is heterogeneous computing system which implements a JVM on top of Tread-Marks distributed shared memory(DSM) system. In DSM system, no explicit communication is necessary and that eliminates the need for the programmer to write message-passing code. Because of the portability feature of Java, the combination of Java and DSM completely hides the hardware differences and the distributed nature of the system from the programmer view, providing a better platform for heterogeneous computing environments.

- **Use pure Java libraries**
  JavaNow [4] is a pure Java based framework on shared memory system that includes features found in Linda, Actors, and Message Passing Interface(MPI). It creates a PVM like virtual parallel machine and provides distributed associative shared memory. JavaNow is currently available as stand-alone software and must be used with a remote (or secure) shell tool in order to run on a network of workstations.

- **Extend Java language with new keywords**
  Titanium [43] is a Java based language and system for high-performance

parallel computing. It has additional features over Java like immutable classes, multidimensional arrays, an explicitly parallel SPMD model of computation with a global address space, and a mechanism for programmer to control memory management. Immutable classes allow the compiler pass objects by value. Because Titanium compiler translates Titanium into C, users do not have full access to the AWT and certain other Java libraries directly from Titanium.

Spar [38] is a programming language for semi-automatic parallel programming, in particular for the programming of array-based applications, supporting multidimensional and sparse arrays. Spar has been designed as part of the Automap project, in which a compiler and run-time system are being developed for distributed memory systems.

javar [6] is a simple Java restructuring compiler, written in C, that can automatically transform a Java program into a form in which implicit parallelism is made explicit by means of multi-threading.

- **Web oriented Java based frameworks**
  WebFlow [22] of Northeast Parallel Architectures Center, Syracuse University, is a particular programming paradigm implemented over a virtual Web accessible metacomputer and given by a dataflow-programming model. WebFlow application is given by a computational visually edited by end-users, using Java applets. Modules are written by module developers, people who have only limited knowledge of the system on which the modules will run. They not need concern themselves with issues such as: allocating and running the modules on various machines, creating connections among the modules, sending and receiving data across these connections, or running several modules concurrently on one machine. The WebFlow system hides these management and coordination functions from the developers, allowing them to concentrate on the modules being developed.

  IceT [19] is an inherently collaborative, multi-user environment. IceT provides mechanism by which resources can be made available to the system for use by users who do not have log-on privileges. The attractive goals of this metacomputing-based approach include better utilization of larger resource bases, as well as an enhanced collaborative environment for high performance computing. Among the challenges introduced by this idea are the numerous security issues introduced by code-upload capabilities for non-privileged uses.

The major categories of Java Grande Related Work are shown in figure 5. We will discuss mpiJava, MPJ and HPJava more in detail next chapter.

Category	Wrapper	Pure Java	Extended Java	Web oriented
System	mpiJava JavaMPI JavaPVM Java/DSM	MPJ MPIJ JPVM JavaNOW	HPJava Titanium Spar javar JavaParty	WebFlow Javelin IceT

Figure 5: Categories of Java Grande Related Work

References

[1] URL: http://www.isye.gatech.edu/chmsr/JavaPVM/.

[2] URL: http://ccc.cs.byu.edu/OnlineDocs/docs/mpij/MPIJ.html.

[3] Java Grande Forum. URL: http://www.javagrande.org.

[4] Shahzad A. Bhatti, Phillip M. Dickens, and George K. Thiruvathukal. Java on Networks of Workstations(JavaNOW): A Parallel Computing Framework Inspired by Linda, Actors, and the Message Passing Interface(MPI). URL: http://www.jhpc.org/projects.htm, http://www.csam.iit.edu/~pmd/pubs.html.

[5] Aart J. C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, June 1997.

[6] Aart J. C. Bik, Juan E. Villacis, and Dennis B. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, November 1997. URL: http://www.extreme.indiana.edu/~ajcbik/JAVAR/.

[7] Fabian Breg, Shridhar Diwan, Juan Villacis, Jayashree Balasubramanian, Esra Akman, and Dennis Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11-13):941–956, sep–nov 1998. URL: http://www.extreme.indiana.edu/hpjava/nexusrmi/.

[8] Bernd O. Christiansen, Peter Cappello, Mihai F. Ionescu, Michael O. Neary, Klaus E. Schauser, and Daniel Wu. Javelin: Internet-based parallel computing using Java. *Concurrency: Practice and Experience*, 9(11):1139–1160, November 1997. URL: http://www.cs.ucsb.edu/research/javelin/.

[9] Kivanc Dincer. jmpi and Performance Instrumentation Analysis and Visualization Tool for jmpi. In *First UK workshop on Java for High Performance Network Computing, Europar '98*, September 1998. URL: http://www.baskent.edu.tr/~kdincer/.

[10] Kivanc Dincer. A Ubiquitous Message Passing Interface Implementation in Java: jmpi. In *Proceedings of IPPS/SPDP 1999(13th International Parallel Processing Symposium and Distributed Processing)*, San Juan, Puerto Rico, April 1999.

[11] J.J. Dongarra, R. Pozo, and D.W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Object Oriented Numerics Conference*, 1993.

[12] Adam J. Ferrari. JPVM: Network Parallel Computing in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Palo Alto, February 1998. URL: http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[13] Stephen J. Fink and Scott B. Baden. The KeLP User's Guide. University of California, San Diego, La Jolla, CA, March 1996. URL: http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html.

[14] High Performanc Fortran Forum. *High Performance Fortran language Specification*. Scientific Programming, special issue 2. 1993.

[15] Ian Foster, George K. Thiruvathukal, and Steven Tuecke. Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System. *Concurrency: Practice and Experience*, 9(6):465–475, June 1997. URL: http://www.globus.org/nexus/nexusjava.html.

[16] Geoffrey C. Fox and Wojtek Furmanski. Computing on the web: new approaches to parallel processing—pataop and exaop performance in the year 2007. 1997. URL: http://www.npac.syr.edu/users/gcf/petastuff/petaweb/.

[17] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Pararallel Virtual Machine—A Users' Guids and Tutorial for Networked Parallel Computing*. Scientific and Engineering Series. MIT Press, 1994.

[18] Vladmimir Getov, Susan Flynn Hummel, and Sava Mintchev. High-performance parallel programming in Java. *Concurrency: Practice and Experience*, 10(11–13):863–872, September 1998.

[19] Paul A. Gray and Vaidy S. Sunderam. IceT: distributed computing and Java. *Concurrency: Practice and Experience*, 9(11):1161–1167, November 1997. URL: http://www.mathcs.emory.edu/icet/.

[20] A.S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report 91 07, University of Virginia, 1991.

[21] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI*. The MIT Press, MIT, Cambridge, Massachusetts, 1994.

[22] Tomasz Haupt, Erol Akarsu, and Geoffrey Fox. WebFlow: a Framework for Web Based Metacomputing. In *High-Performance Computing and Networking '99*, Amsterdam, April 1999.

[23] George Crawford III, Yoginder Dandass, and Anthony Skjellum. The JMPI Commercial Message Passing Environment and Specification: Requirements, Design, Motivations, Strategies, and Target Users. Technical report, MPI Software Technology, Inc, 1998.

[24] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed Object Group Management Architecture. In *ACM 1998 Workshop on Java for High-performance Network Computing*, volume 10(11-13), Palo Alto, February 1998. URL: http://ccc.cs.byu.edu/DOGMA/.

[25] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, and Brad Topol. Efficient implementation of Java Remote Method Invocatio(RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and System(COOTS'98)*, 1998.

[26] Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Henri E. Bal, and Aske Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN symposium on principles and practice of parallel programming(PPoPP'99)*, pages 173–182, May 1999. URL: http://www.cs.vu.nl/manta/.

[27] Message Passing Interface Forum, University of Tenessee, Knoxville, TN. *MPI: A Message-Passing Interface Standard*, June 1995. URL: http://www.mcs.anl.gov/mpi.

[28] Sava Mintchev and Valdimir Getov. Towards Portable Message Passing in Java: Binding MPI. In *Proceedings of EuroPVM-MPI'97*, pages 135–142, Kraków, Poland, November 1997. Springer LNCS 1332.

[29] Christian Nester, Michael Philippsen, and Bernahard Haumacher. A more efficient RMI for Java. In *Proc. ACM 1999 Java Grande Conference*, pages 152–159, San Francisco, June 1999.

[30] Martin Odersky and Michael Philippsen. Espresso grinder, 1996. URL: http://ipd.ira.uka.de/~espresso.

[31] Martin Odersky and Philip Walder. Pizza into Java: Translating theory into practice. In *24th ACM Symposium on Principles of Programming Languages*, January 1997. URL: http://ipd.ira.uka.de/~pizza.

[32] Michael Philippsen and Bernhard Haumacher. Bandwidth, Latency, and other Problems of RMI and Serialization. In *Second Java Grande Forum Meeting*, Palo Alto, May 1998. URL: http://www.javagrande.org/meetings.htm.

[33] Michael Philippsen and Bernhard Haumacher. More Efficient Object Serialization. In *Parallel and Distributed Processing*, LNCS 1586, pages 718–732. International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, April 1999.

[34] Michael Philippsen and Matthias Zenger. JavaParty-Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. URL: http://wwwipd.ira.uka.de/JavaParty/.

[35] Rajeev R. Raje, Joseph I. William, and Michael Boyles. An Asynchronous Remote Method Invocation(ARMI) Mechanism for Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, November 1997.

[36] George K. Thiruvathukal, Lovely S. Thomas, and Andy T. Korczynski. Reflective remote method invocation. In *Concurrency: Practice and Experience*, volume 10(11-13), pages 911–926, September-November 1998. URL: http://www.jhpc.org/projects.html.

[37] Rob van Nieuwpoort, Jason Maassen, Henri E. Bal, Thilo Kielmann, and Ronald Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, California, June 1999. URL: http://www.cs.vu.nl/manta/.

[38] Kees van Reeuwijk, Arjan J.C van Gemund, and Henk J. Sips. Spar: a programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997. URL: http://www.pds.twi.tudelf.nl/~reeuwijk/projects/javaws_poster.html.

[39] Matt Welsh. Ninja. URL: http://www.cs.berkeley.edu/~mdw/proj/ninja/.

[40] Matt Welsh. NinjaRMI. URL: http://www.cs.berkeley.edu/~mdw/proj/ninja/ninjarmi.html.

[41] Gregory V. Wilson and Paul Lu. *Parallel Programming using C++*. MIT, 1996.

[42] Narendar Yalamanchilli and William Cohen. Communication Performance of Java based Parallel Virtual Machines. In *ACM 1998 Workshop on Java for high-Performance Network Computing*, Palo Alto, February 1998.

[43] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance network computing*, February 1998. URL: http://www.cs.berkeley.edu/Research/Projects/titanium/.