Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

A Comparison of High Performance, Parallel Computing Java Packages

Nan C. SCHALLER and Sidney W. MARSHALL
Computer Science Dept., Rochester Institute of Technology
102 Lomb Memorial Drive, Rochester, NY 14623-5608, USA
ncs@cs.rit.edu, swm@cs.rit.edu

Yu-Fong CHO
R&D Group, Askey Computer Corp.
10F, NO 119, Chien-Kang Road, Chung-Ho, Taipei 235, Taiwan
yfcho@askey.com.tw

Abstract. The high-performance computing community has developed numerous Java packages that support parallel and distributed computing. Most of these packages are designed for the typical parallel message passing and shared memory architectural paradigms. This paper presents the results of a recent study that included a web search for such packages, describes the paradigms implemented in them, and evaluates their performance on a parallel, 4-processor SMP machine using three benchmark programs that represent a mix of typical parallel applications, chosen from *The Java Grande Benchmark Suite.* A brief description of each package and a discussion its ease of installation and use are also provided.

1 Introduction

Today, the high-performance computing (HPC) community is more interested than ever in the possibility of using Java. It is becoming more viable to do so. As Pancake and Lengauer recently reported: "Previous Java implementations focused mainly on the portability and interoperability aspects required for Internet-centric client/server computing. Because Java was originally interpreted, it is commonly perceived as being execution inefficient. Recent developments in compiler technology and instruction-level optimisations have done away with many of the sources of this inefficiency. Java's recent execution efficiency improvement is due to static analysis, just-in-time compilation and optimization of the Java Virtual Machine (JVM). Software scientists are also making efforts to improve Java's performance of remote method invocation (RMI), numeric capabilities, and communication mechanisms. These days, Java is competitive with C and C++ for some applications on some platforms, and it is considerably safer to execute and easier to program." [1]

This paper reports on a study that evaluated several publicly available Java packages for HPC with respect to ease of installation, ease of use, and performance. Some of these packages were sponsored by the *Java Grande Forum* [2], a major consortium representing the interests of the Java high-performance computing community. The Forum's benchmark software was used to measure and compare the performance of these packages.

2 Background

Parallel computing platforms are usually either single computers with multiple internal processors or multiple interconnected computers. Two methods for communicating between such processors use message passing or shared memory. In message passing systems, memory is distributed among the processors, each processor having its own address space. In this case, a processor can only directly access its own memory and an interconnection network is necessary for processors to be able to send messages to other processors. Shared memory multiprocessors use a single address space. Here, each location in memory has a unique address that may be used by each processor to access that location. [3]

The HPC community has developed several Java packages that are suitable for these architectural paradigms. For example, for message passing, there are packages based on the Communicating Sequential Processes (CSP) [4], Message Passing Interface (MPI) [5], and Parallel Virtual Machine (PVM) [6] models and for shared memory, there are packages based on the Open Multi Processing (OpenMP) [7] and Linda [8] models. Brief descriptions of these models are provided below.

- **Message Passing using CSP**
  "Communicating Sequential Processes (CSP) is a mathematical theory for specifying and verifying the complex patterns of behavior arising from interactions between concurrent objects. CSP has a formal, and compositional semantics that … encapsulates fundamental principles of processes, networks and communication." [9] CSP programs are written using *processes*, *networks* of processes and various forms of *synchronization* and *communication* between them.
- **Message Passing using MPI**
  "MPI (Message Passing Interface) is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementers, and users. The message-passing application programmer interface (API) is combined with protocol and semantic specifications for how its features must behave in any implementation. MPI includes point-to-point message passing and collective operations." [5]
- **Message Passing using PVM**
  "PVM (Parallel Virtual Machine) is an integrated set of software tools and libraries that permits a heterogeneous collection of computers hooked together by a network to be used as a parallel computer." [6][10][11]
- **Shared Memory using OpenMP**
  "OpenMP (Open Multi Processing) is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism. The OpenMP Application Program Interface supports multi-platform shared-memory parallel programming on all architectures..." [7]
- **Shared Memory using Linda**
  Linda is a concurrent programming model whose primary concept is that of a `tuple-space`, an abstraction via which cooperating processes communicate ... Linda … provides a shared-memory abstraction for process communication without requiring the underlying hardware to physically share memory." [8][12]

3 A Survey of Packages Found

The purpose of this study was to locate HPC Java packages on the Web, and evaluate their ease of installation, their ease of use, and their performance. After locating many such packages, the study was narrowed to those packages that

- Supported either the message passing or shared memory programming models.
- Were publicly available, i.e., not commercial software.
- Worked with the standard Java Development Kit (JDK).
- Were up-to-date. (For the purposes of this study, JDK 1.4.0 was considered up-to-date.)

The Web search was performed in April 2002 with the evaluation process continuing through September 2002. Packages found under these narrowed criteria are listed below along with a brief description taken from each web site, and are summarized in Table 1.

3.1 Message Passing using CSP

Two Java packages were found that use the CSP model: Communicating Threads for Java (*CTJ*) [13] and Communication Sequential Processes for Java (*JCSP*)[14]. As both *CTJ* and *JCSP* were implemented using the CSP model, they have many concepts in common[15]. An important difference between them is their process scheduling systems.

CTJ's process scheduling kernel is specially designed for programming small, real-time embedded systems. [16][17] "The prototype *CTJ* package provides the thread/CSP model of processes, channels, and composition constructs for Java… It implements the CSP model for the standard Java monitor/threads operations, and enables any Java threaded system to be analyzed in CSP terms." [16]

JCSP also implements the CSP model, but relies on the Java Virtual Machine (JVM) for its thread scheduling. "*JCSP* is a 100% Java class library providing a base range of CSP primitives. It also includes a package providing CSP process wrappers giving a channel interface to all Java AWT widgets and graphics operations." [14]

3.2 Message Passing using MPI

Two packages were found that use the MPI model: *JavaMPI* [18] and *mpiJava* [19].

"*JavaMPI* is a Java binding for MPI... The *JavaMPI* library contains an MPI library that is dynamically linked to the JVM during program execution." [20]

"*mpiJava* is an object-oriented Java interface to standard MPI. *mpiJava* does not assume any special extensions to the Java language. It can be ported to any platform that provides compatible Java development and native MPI environments. The *mpiJava* package is implemented as a set of Java Native Interface (JNI) wrappers to native MPI packages. Platforms currently supported include Solaris using MPICH [21] or SunHPC-MPI [22], Linux using MPICH [21], and Windows NT using WMPI [23]." [19] The *mpiJava* package was developed as part of the *HPJava* project [24] supported by the Northeast Parallel Architectures Center (NPAC) [25] at Syracuse University.

3.3 Message Passing using PVM

The web search found two Java packages that use the PVM model: *jPVM* (JavaPVM) [26] and *JPVM* [27].

"*jPVM* is an interface written using the Java native methods capability that allows Java

applications to use the native Parallel Virtual Machine (PVM) [6] software developed at Oak Ridge National Laboratory. *jPVM* extends the capabilities of PVM to the Java architecture-independent programming language. *jPVM* allows Java applications and existing C, C++, and Fortran applications to communicate with one another using the PVM API. " [26]

"*JPVM* is a PVM-like library of object classes implemented in and for use with the Java programming language. It combines two advantages, ease of programming inherited from Java and high performance through parallelism inherited from PVM." [27]

3.4 Shared Memory using OpenMP

The only Java package found using the OpenMP model was *JOMP* [28].

"*JOMP* implements the OpenMP Application Program Interface (API) in Java. It provides OpenMP-like directives and methods consisting of a compiler and runtime library written entirely in Java. *JOMP* implements most of the OpenMP specification.

JOMP uses the fork-join model of parallel execution. A program written using the *JOMP* API begins execution on a single thread called the master thread. The master thread executes in a serial region until the first parallel construct is encountered, whereupon the master thread creates a team of threads, which includes itself. Each thread then executes the code in the dynamic extent of the parallel region..." [29][30]

3.5 Shared Memory using Linda

Two packages were found that used the Linda model: *Jada* [31] and *JavaSpaces* [32].

"*Jada* adds operations to access Linda-like multiple tuple-spaces. *Jada*'s design goal was simplicity rather than performance. *Jada*, like Linda, is a minimalist coordination language. Other Linda-like implementations usually include a pre-processor, necessary because Linda slightly changes the host language syntax. *Jada* is based on a set of classes that are used to access a tuple-space... This allows users to use their standard Java development tools. *Jada* is implemented as a set of classes that allow either Java threads or Java applications to access an associatively shared tuple space using a small set of Linda-like operations." [33][31]

"*JavaSpaces* technology is a simple unified mechanism for dynamic communication, coordination, and sharing of objects between Java technology-based network resources like clients and servers. In a distributed application, *JavaSpaces* technology acts as a virtual space between providers and requesters of network resources or objects. This allows participants in a distributed solution to exchange tasks, requests and information in the form of Java technology-based objects... The design of *JavaSpaces* was strongly influenced by Linda. *JavaSpaces* systems are similar to Linda systems in that they store collections of information for future computation and are driven by value-based lookup." [34] *JavaSpaces* technology was developed by Sun Microsystems [35] and is included in the Jini package.

3.6 Others

Other Java solutions for high performance computing were found. Some of them were replacements for the Java compiler, or the JVM. Furthermore, some of them replaced the Java language itself, using an extended Java syntax to gain the benefits of Java. These were not included in the study.

3.7 Summary of Results

Table 1 provides a summary of the Java packages found during the web search that met the criteria set forth. It lists the JDK version used for development, the programming paradigm, the version tested, and the date of latest update at the time of the study.

Package Name	JDK Version	Programming Paradigm	Version tested	Latest Update
CTJ	Not specified	Message passing using CSP	0.9, r18	October 2000
JCSP	1.1.5 or later	Message passing using CSP	1.0-rc4	Feb. 2002
JavaMPI	Not specified	Message passing using MPI	0.4	November 1998
mpiJava	1.2 or later	Message passing using MPI	1.2.3	October 2001
jPVM	1.1.5 or later	Message passing using PVM	1.1.4	April 1998
JPVM	Not specified	Message passing using PVM	0.2	Feb. 1999
Jada	Not specified	Shared memory using Linda	3.0 beta	April 2001
JavaSpaces	1.2.2_007 or later	Shared memory using Linda	1.2.1	April 2002
JOMP	1.2 or later	Shared memory using OpenMP	1.0 beta	2000

Table1: Summary of Java Packages

4 Installation and Usage

For the purpose of this study, all packages were installed and tested on a Sun Microsystems Enterprise 450 running the Solaris 8 Operating System. This SMP system has four UltraSPARC II 248 MHz CPUs with 2048 Megabytes system memory, and a system clock frequency of 83 MHz. As some of the packages were designed for an SMP system and could not be used on a cluster or distributed systems, testing was limited to this single four-processor system. All benchmark programs were compiled using the standard JDK compiler (`javac`), and were executed using the standard JDK Java virtual machine (`java`).

In several cases, all that was needed to set up the package was to download it, uncompress a file or two, and add the appropriate directory to the `CLASSPATH` environment variable. This was the case for *CTJ*, *JCSP*, *Jada* and *JOMP*. For others, it was more complicated. And, some packages could not be installed on the system. The following subsections contain notes regarding installation and programming with these packages. Table 2 summarizes this information.

4.1 CTJ

Besides needing to understand CSP to program with *CTJ*, special effort is required to run in parallel. The terms "*Thread*" in Java and "*Process*" in *CTJ* are closely related, but do not act the same way. *CTJ* does not perform time slicing by default, even if a programmer creates a multi-process program. This means that *CTJ* does not context-switch between processes unless those processes engage in an event such as communicating with each other through a *CTJ* channel. *CTJ* does provide a `TimeSlicer` class to enable time slicing among a single processor's subprocesses. *CTJ* has its own process scheduling kernel, and is specially designed for small, embedded, real-time systems. In addition, *CTJ* does not automatically distribute processes across processors. Therefore, to run in parallel, the user must manually start a process on each processor. In addition, special `channel` classes must be provided to implement communication on a distributed memory system.

4.2 JCSP

Programming under *JCSP* is similar to programming using Java threads. Because it is based on CSP, *JCSP* purports to provide a superior mechanism to the Java threads mechanism for handling multi-threaded programming. A program unit is called a *process* rather than a *thread*. Programmers can either simply use the *JCSP* process mechanism to replace the Java threads or use the CSP model to implement parallel programs. While, *JCSP* was designed to be efficient on any parallel architecture, the public domain JCSP version tested only runs in parallel on shared memory systems.. The user must implement classes to provide communication on distributed memory systems or purchase the commercial version. (*JCSP* was commercially released by Quickstone Technologies [36] during the time of our study.)

4.3 mpiJava

A native MPI C interface, such as MPICH [21] or SunHPC MPI [22], is required to use the *mpiJava* package. The default interface is MPICH. Our attempts to modify the *mpiJava* startup script to work with SunHPC 3.1, the MPI version already installed on the test system, were not successful. Therefore, MPICH was used instead. It should be noted that we were successful with SunHPC 4.0 later on a different system.

The installation and usage of *mpiJava* requires precaution, especially if, as in our case, the system uses Secure Shell (SSH) services, i.e., telnet services are not available. Such information must be provided before compiling and installing both MPICH and *mpiJava*. In addition, the SSH agent must be set up to automatically log on without requiring a password before running *mpiJava* programs.

However, programming under the *mpiJava* environment is straightforward. *mpiJava* provides wrapper classes for the MPI package as extended Java classes, so using an MPI function is the same using any Java class. And, it is similar to other MPI programming environments.

4.4 Jada

Programming *Jada* is straightforward. *Jada* implements a Linda-like `tuplespace` called `ObjectSpace`. Linda-like programming models use the term *worker* to represent a thread/process. Communication takes place between workers by depositing `tuples` into and withdrawing `tuples` from `ObjectSpace`.

4.5 JavaSpaces

In order to use *JavaSpaces*, users must install the Jini package from Sun Microsystems, Inc. [35]. Installation is easy, but starting *JavaSpaces* is much more complicated, since Jini must be started first. This requires that the following services be started: an `HTTP Server`, an `RMI Activation Daemon`, an `RMI Registry` or a `Jini Lookup service`, a `Transaction Manager`, and then finally *`JavaSpaces`*. In addition, a Java security policy must be defined before execution. It is complicated to get everything started correctly.

JavaSpaces uses an `Entry` class that acts like a Linda-like tuple, i.e., to be written into and to be taken out of *JavaSpaces*. Each field of an entry must be a public object type, and an `Entry` cannot store primitive types in its fields. Programmers who are just starting to program using *JavaSpaces* technology could easily miss this restriction.

4.6 JOMP

Programming in the *JOMP* environment is similar to programming in other OpenMP programming environments. *JOMP* provides OpenMP-like syntax and programming style. It also provides a *JOMP* pre-compiler that will translate a Java program into a *JOMP* parallel program, which helps programmers focus on their parallel algorithm and not *JOMP* syntax.

4.7 Packages Not Tested

The *JPVM*, *jPVM* and *JavaMPI* packages, mentioned in the previous section, were not tested. Research had ended for all three packages and their most recent implementations would not execute under the JDK specified in our refined criteria. Furthermore, MPI seems to have, for the most part, replaced PVM as the de facto message passing standard.

Package Name	Additional Required Software	Ease of Installation	Ease of Use	Remark
CTJ	None	Easy	Need knowledge of CSP	
JCSP	None	Easy	Need knowledge of CSP	
JavaMPI	MPI/LAM	Not compatible with hardware	Out-of-date	Not tested
mpiJava	MPICH or SUN HPC	Need to compile both packages	Need knowledge of MPI	
jPVM	PVM		Out-of-date	Not tested
JPVM	None		Out-of-date	Not tested
Jada	None	Easy	Need knowledge of Linda	
JavaSpaces	Jini	Complicated	Need knowledge of Linda and complicated to set up.	
JOMP	None	Easy	Easy	

Table 2: Ease of Installation and Use

5 Benchmarks

The purpose of benchmark testing is to provide a means to meaningfully measure and compare alternative execution environments. Three benchmarks were chosen for this study out of the five available in *The Java Grande Forum Benchmark Suite[37]*. "These algorithms are designed to use large amounts of processing, I/O, network bandwidth, or memory." [37] Thus, each algorithm chosen represents a particular type of application program. "These codes are simple kernels that reflect the type of computation that can be expected to be found in the most computation intense parts of real numerical applications"[38]. However, they are not the optimal implementations of these algorithms. Brief descriptions of the benchmark algorithms chosen follow.

5.1 The Crypt Benchmark

"The Crypt benchmark performs International Data Encryption Algorithm (IDEA) encryption and decryption of an array of *N* bytes. This algorithm involves two principal loops, whose iterations are independent and may be divided between processors in a block fashion." [38] The Crypt benchmark represents parallel applications that are both computation and communication intensive. The parallel version of this benchmark distributes encryption computation to processors, merges the results of encryption,

distributes decryption computation to processors, and then collects the results of decryption. An *N* byte array must be transmitted between processors whenever communication is needed. Synchronization is required at the end of the encryption and decryption steps.

5.2 The Series Benchmark

"The Series benchmark computes the first *N* Fourier coefficients of the function $f(x)=(x+1)^x$ on the interval [0, 2]. The most time consuming component of the benchmark is the loop over the Fourier coefficients. The calculation of each coefficient is independent of every other coefficient and the work may be distributed simply between processors." [38] The Series benchmark represents the purely computation intensive parallel application. The parallel version of this benchmark distributes the computation to the processors and then collects the computed Fourier coefficients. Little communication is required. Synchronization is required at the beginning and the end of the benchmark.

5.3 The SOR Benchmark

"The SOR benchmark performs 100 iterations of Successive Over-Relaxation (SOR) on an *N x N* grid. This benchmark involves an outer loop over iterations and two inner loops, each looping over the grid. In order to update elements of the principal array during each iteration, neighbouring elements of the array are required, including elements previously updated. Hence this benchmark is inherently serial. To allow parallelism to be carried out, the algorithm has been modified to use a "*red-black*" ordering mechanism. This allows the loop over array rows to be parallelised. Hence, the outer loop over elements has been distributed between processors by columns." [38]

The SOR Benchmark represents the most communication intensive application of these three benchmarks. In the parallel version, an *N x N* grid must be distributed equally to the participating processors at the beginning of the program, *2N* rows of the grid must be exchanged between processors for each iteration, and the *N x N* grid must be returned to the host processor at the end of the program. Synchronization occurs at the beginning of the program, at the end of each iteration, and at the end of the benchmark.

6 Performance Results

The benchmarks were executed for a variety of dataset sizes on the test platform. It should be noted that the figures that follow show the results for the largest datasets only for each benchmark. The raw data is provided as well in tabular form in Appendix A. Other figures are available from the authors upon request. The execution of the sequential version of the benchmark was used as the basis for speedup calculations. However, as these sequential versions are not purported to be the "best" algorithms, it might be appropriate to consider the speedup measurements below as "relative" rather than "theoretical". Benchmark results are expected to improve as the dataset size increases. In addition, communication overhead is expected degrade performance. However, bigger data sets requiring more computation can be expected to mitigate some of this degradation.

This study utilized a set of Java timing utility classes, available from The *Java Grande Forum Benchmark Suite [37]*. All of the times measured were actual algorithm computation times in seconds excluding initialization and I/O time. Each benchmark test was executed ten times to evaluate the variation of the results; there was little variance from test to test. Thus, an average execution time was used.

Execution time should ideally decrease as the number of processors used increases.

Speedup is calculated using formula 6.1 and indicates how much faster the parallel version is than the sequential version. Ideally, speedup should increase linearly as the number of processors increases. Computation intensive algorithms are expected to perform better, i.e., exhibit more speedup, than communication intensive algorithms.

$$Speedup = \frac{ExecutionTimeOfSequentialVersion}{ExecutionTimeOfMutiProcessorsVersion} \quad (6.1)$$

6.1 Results for the Crypt Benchmark

The Crypt benchmark represents applications that are both computation and communication intensive. Benchmarks were run using datasets of 3 MB, 20 MB, and 50 MB arrays. Figure 1 shows the execution time of all packages for this benchmark for the 50 MB dataset and Figure 2 shows the speedup. These figures show that most packages performed well, with the exceptions of *JavaSpaces*, *mpiJava* and *CTJ*.

JavaSpaces performed worse than any other package for this benchmark. However, *JavaSpaces* did show some benefit from a multiprocessor environment. As mentioned above, Jini services, which consume a lot of system resources, must be started correctly before starting *JavaSpaces*. For example, almost 300 MB of system memory was already consumed before benchmark programs were loaded. Furthermore, *JavaSpaces* threw an `OutOfMemory` exception for the 20 MB or 50 MB datasets. Assigning larger memory to the JVM did not solve this problem.

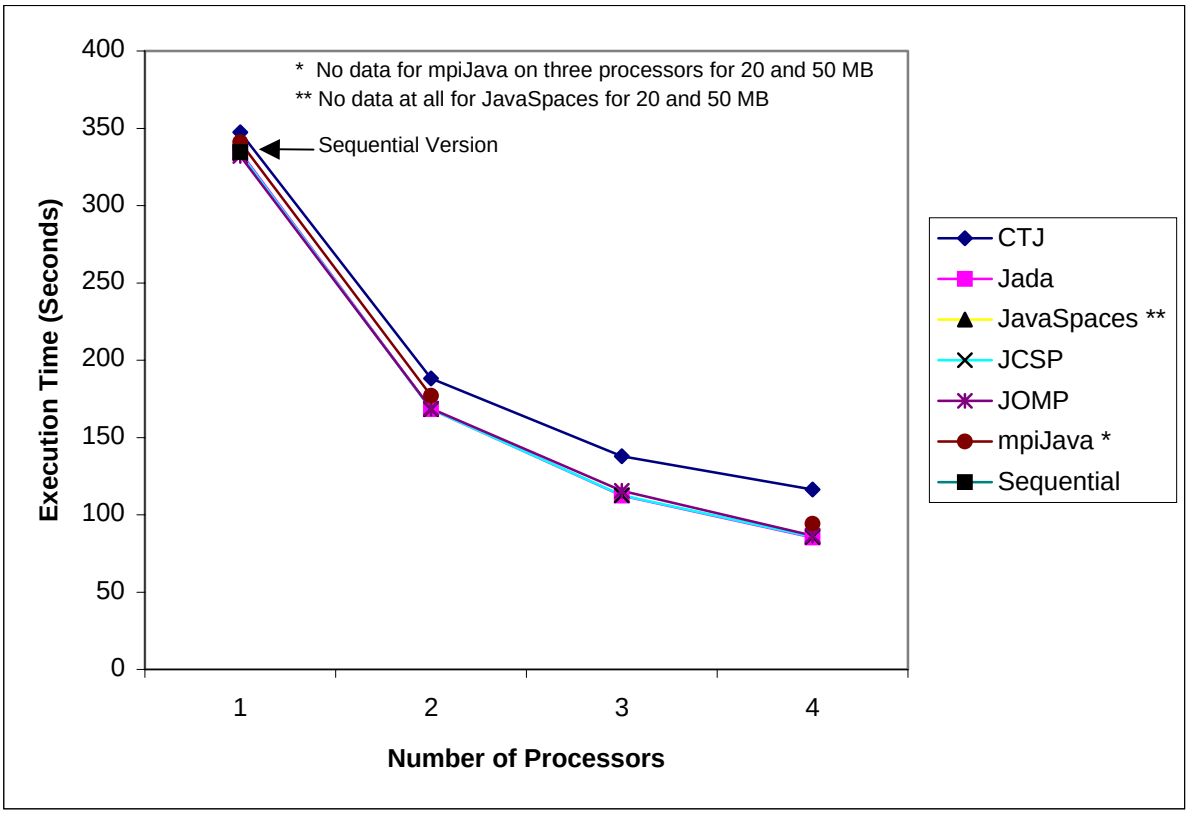

Figure 1. Execution time of Crypt benchmark for 50 MB array dataset

It is interesting to note that the other Linda-like package, *Jada*, performed well for the Crypt benchmark. Although both of *JavaSpaces* and *Jada* are implemented as Linda-like

shared memory paradigm, their implementations are different. *Jada*'s implementation takes the advantage of the SMP system, but *JavaSpaces* creates a virtual memory over the network of computers. *JavaSpaces* then uses network-based communication even when shared memory is available. This type of communication is not as fast as but does have better scalability.

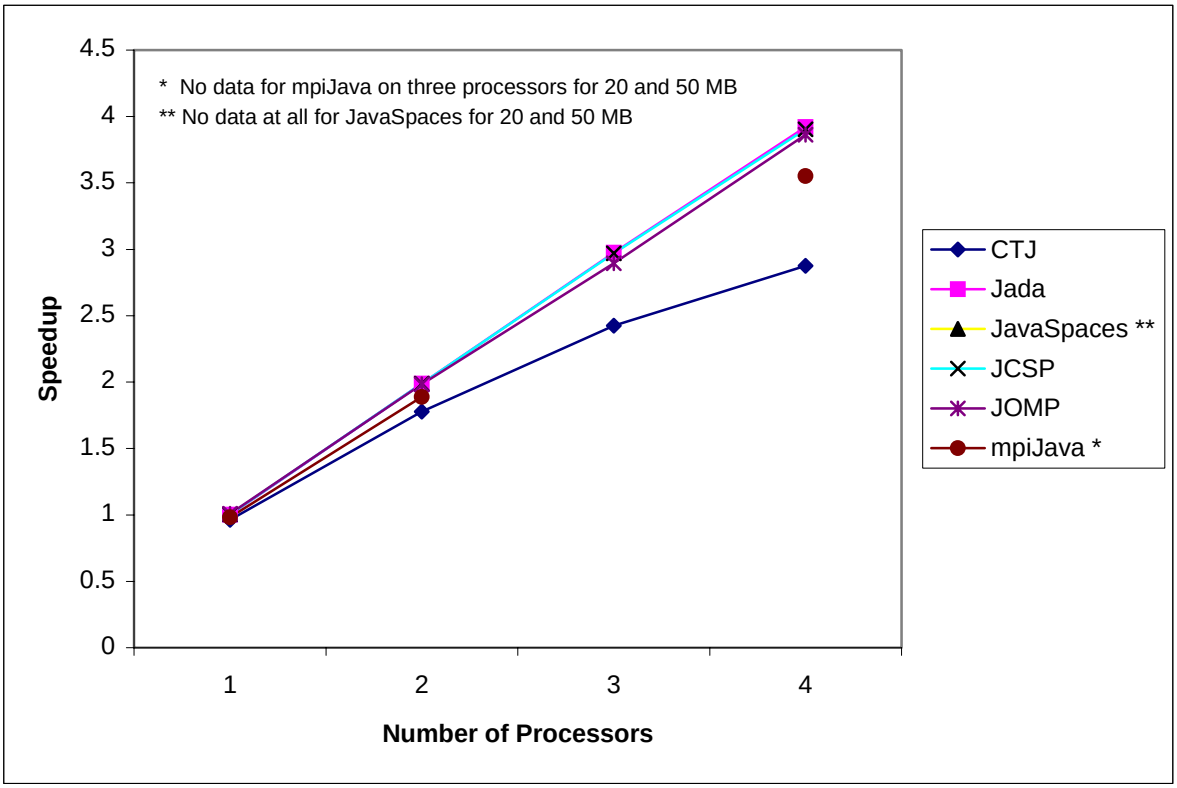

Figure 2. Speedup for Crypt benchmark with 50 Megabyte array dataset

CTJ performed much better than *JavaSpaces* but not as well as the other packages. It was slower, particularly when the number of processors used was increased. This is likely due to the mechanism used to handle communication between individual processors.

We also experienced difficulty with *mpiJava* when using three processors for the 20 MB and 50 MB datasets. We have not been able to identify the cause of the problem, but the MPICH FAQ indicated that it might be due to compiler implementation incompatibility. A new version of MPICH (1.2.4) was released at end of May 2002. More investigation is required to see if this problem is resolved with this newer version, or by utilizing SunHPC's MPI.

To summarize, the Crypt benchmark represented common applications that are both computation and communication intensive. All packages showed reasonably good performance with the exception of *JavaSpaces*, *mpiJava* and *CTJ*. When no other issues arose, as expected, performance was better the larger the dataset.

6.2 Results for the Series Benchmark

The Series benchmark represents a computation intensive application. Dataset sizes were limited to 10K and 100K Fourier coefficients as the time to process a 1000K Fourier coefficient dataset proved prohibitive. Figure 3 shows the execution time of all packages

for the Series benchmark calculating 100K Fourier coefficients, while Figure 4 shows the speedup. As is shown in these figures, the performance of all packages was roughly the same. This is because of the benchmark's computation intensive nature, i.e., not much communication is needed. Interestingly, a few of the single processor versions, most notably *JCSP*'s, did execute faster than the sequential version.

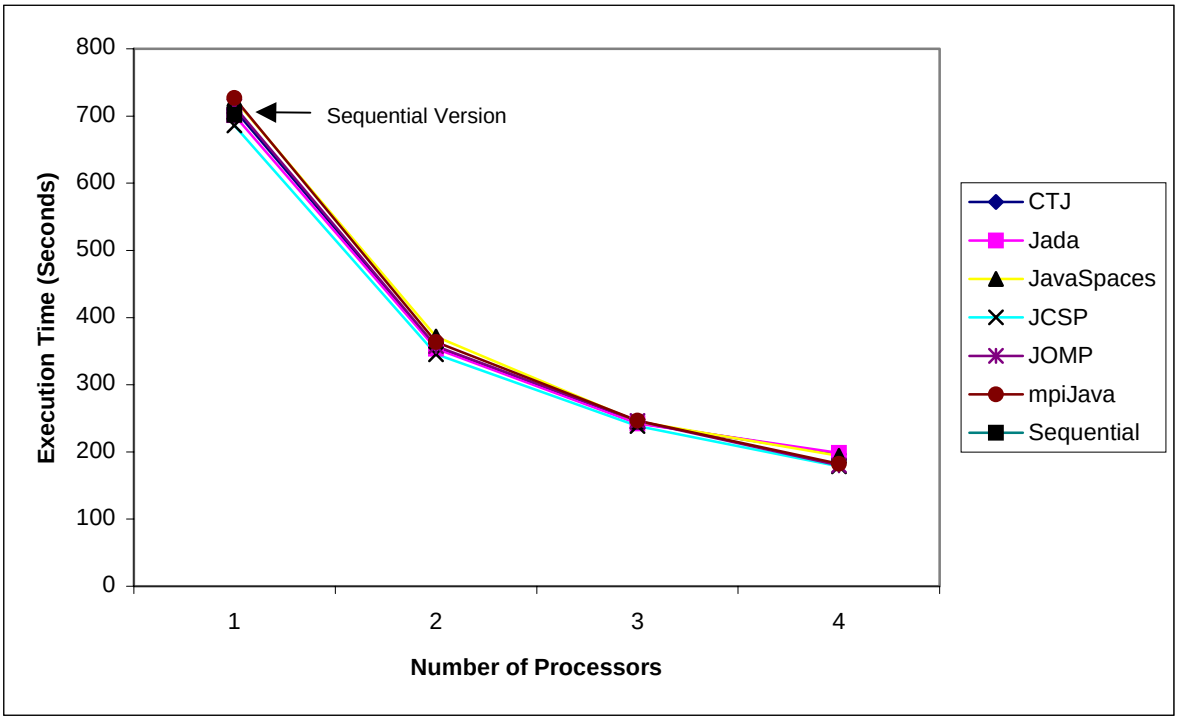

Figure 3. Execution Time of Series benchmark for 100K Fourier coefficients

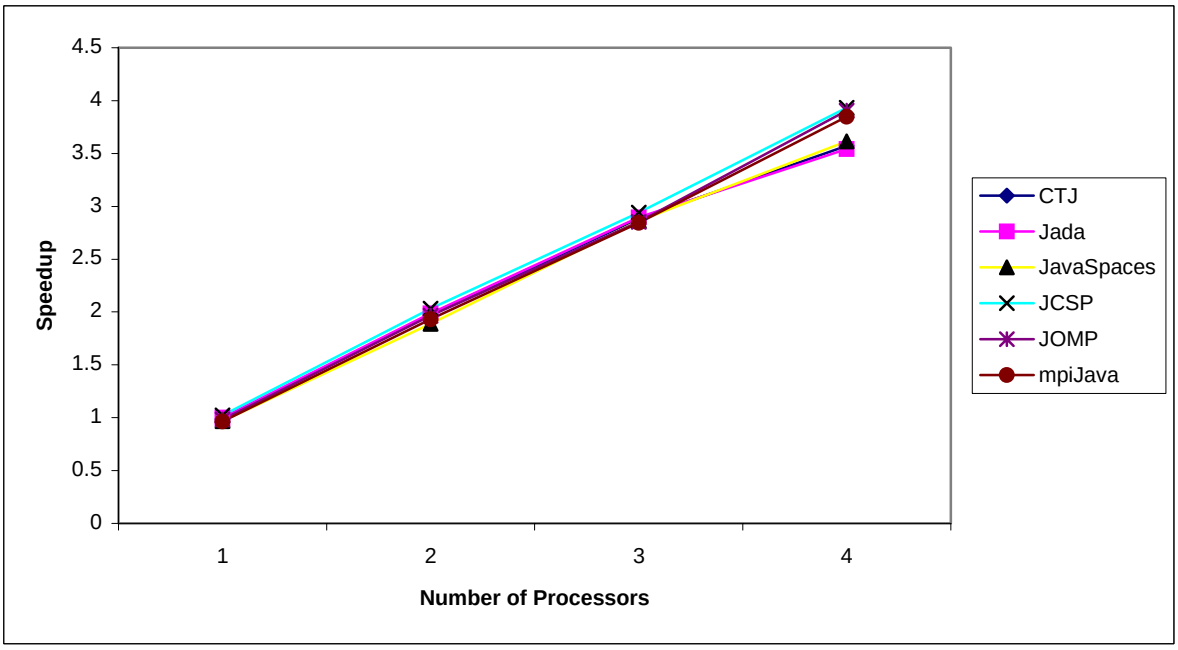

Figure 4. Speedup for Series benchmark for 100K Fourier coefficients

The results varied most when four processors were used. *CTJ*, *JavaSpaces* and *Jada* all deviated more from the ideal speedup than other packages in this situation. All other packages showed near linear speedup. We conjecture that a larger dataset might alleviate this deviation.

In summary, the Series benchmark represented the purely computation intensive application. Packages performed most consistently for this benchmark. As expected, the larger the dataset, the better the performance.

6.3 Results for the SOR Benchmark

The SOR Benchmark was the most communication intensive of the benchmarks used. Datasets of grid sizes *1000 x 1000*, *1500 x 1500*, and *2000 x 2000* were used. Figure 5 shows the execution time of all packages for the *2000 x 2000* grid and Figure 6 shows the speedup. Package performance varied more widely for this benchmark than for the other two, demonstrating how communication overhead can undermine the benefit of adding more processors.

The single processor versions were slower than the sequential version for all grid sizes and packages, but most packages demonstrated some speedup when using more than one processor. Performance was better for larger datasets for all packages.

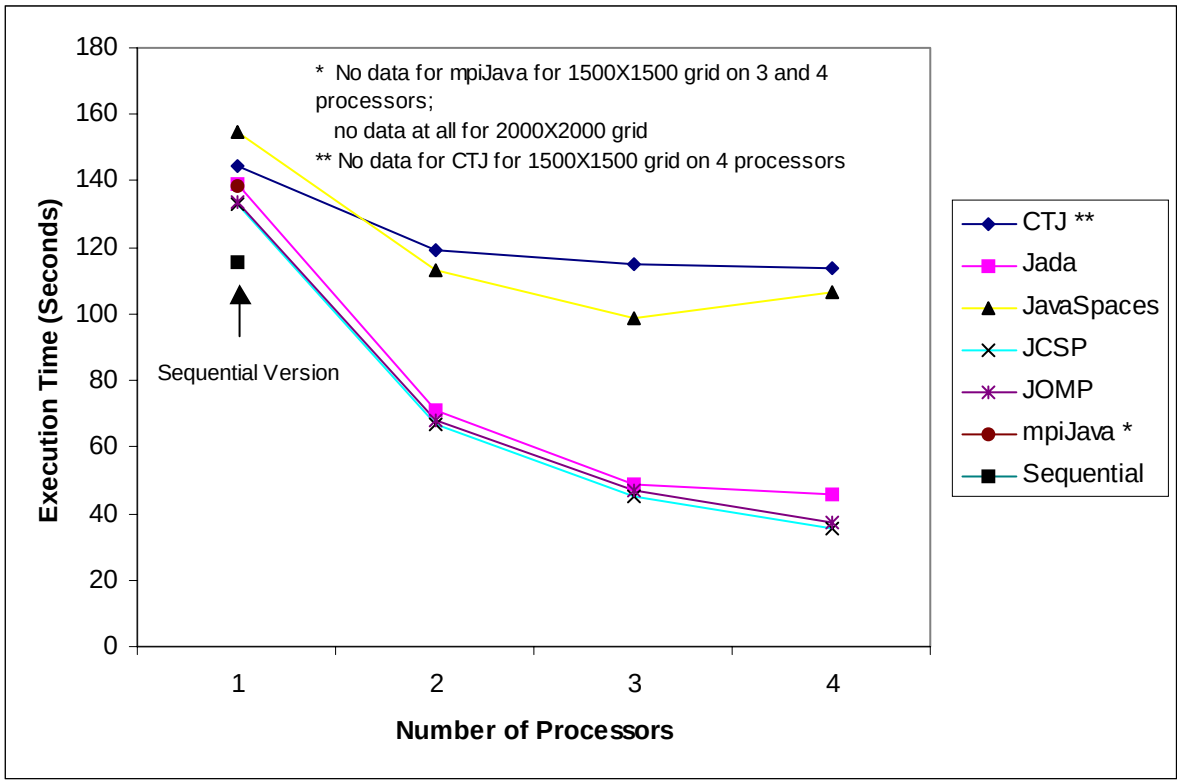

Figure 5. Execution Time of SOR benchmark for a *2000 x 2000* grid

While *CTJ* and *JavaSpaces* did benefit from the multiprocessor environment with the two smaller grids, the more processors used, the worse their performance. In fact, all execution times were slower than that of the sequential version. This is caused by heavy communication requirements along with the way in which communication is implemented for these packages. In addition, we unable to run the *1500 x 1500* grid for *CTJ* with four processors and have not yet determined the cause.

The performance of *JavaSpaces* and *CTJ* improved with the *2000 x 2000* grid. *JavaSpaces*' execution times were faster, in this case, than for the sequential version. This shows that, as expected, communication overhead decreases as the problem size increases.

We were unable to obtain results using *mpiJava* on three and four processors with the *1500 x 1500* grid and obtained no results at all for the *2000 x 2000* grid. We received error messages similar to the ones received while running the Crypt benchmark.

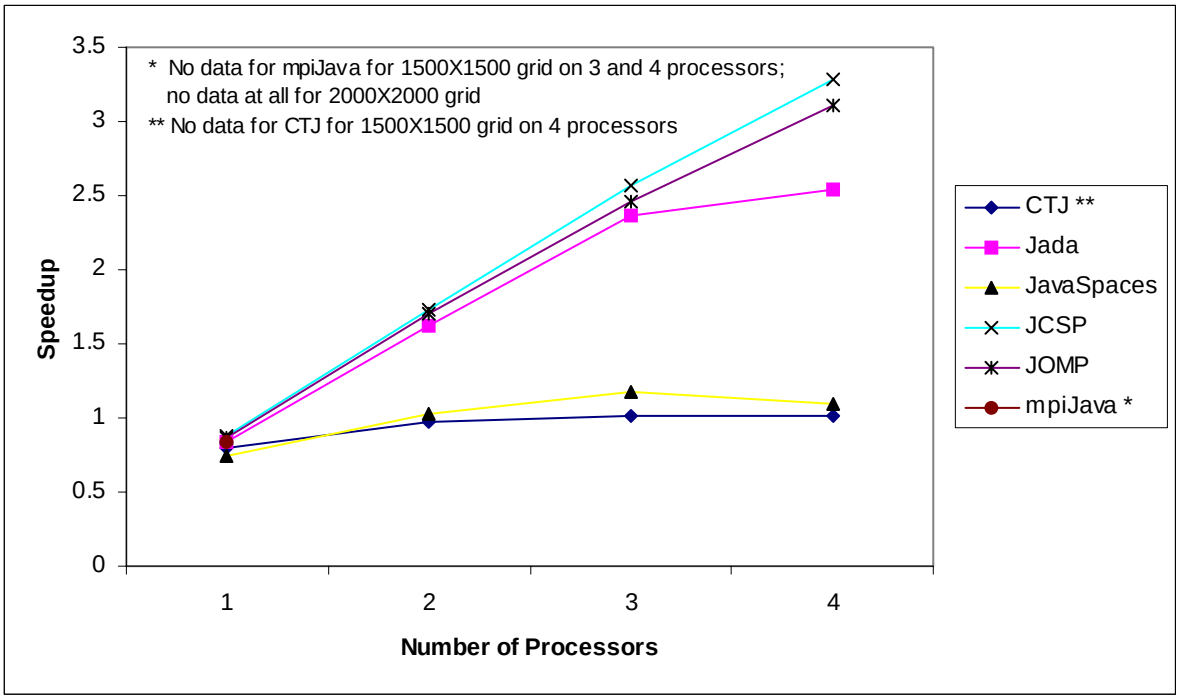

Figure 6. Speedup for SOR benchmark for a *2000 x 2000* grid

Jada performed better when handling larger datasets, but did not show increased speedup when adding a fourth processor for *1000 x 1000* and *1500 x 1500* grids. However, it did run faster on four processors than on one processor or two processors, and it performed much better than the other Linda-like package, *JavaSpaces*.

In summary, the performance of all packages for the SOR benchmark was, as expected, not as good as they were for the Crypt and Series benchmarks. The SOR Benchmark performs more communication operations than either of the other benchmarks, and it performs many bulk synchronization operations. These two factors account for the performance differences. However, most of the packages did perform better with larger dataset sizes.

7 Limitations and Future Work

This study was limited to high-performance Java packages that were up-to-date, non-commercial, freely available, and able to execute using JDK 1.4.0 on the test platform. The packages were evaluated on an SMP system only. While *JOMP* operates only in a shared memory, the other packages could, with some work, be evaluated on distributed memory environments as well. In some cases, the modifications necessary to do this are relatively simple. For example, only the configuration file need be modified for *mpiJava*. In others, it is more difficult. For example, the communication mechanism for *Jada*, *CTJ*, and *JCSP* must be modified or replaced. The commercial version of JCSP from Quickstone[36] does provide this.

Future work will focus on

- Extending the study to include the other benchmarks available in *The Java Grande Forum Benchmark Suite* [37].
- Examining a wider range of platforms, Java environments, and packages. For example, the benchmarks for each packages, when appropriate, could be evaluated on a cluster.
- Evaluating additional packages and newer versions of these packages, i.e., those updated after April 2002.
- Using a variety of different implementations of the auxiliary software. For example, evaluating *mpiJava* with MPICH 1.2.4 and SunHPC MPI 5.0, rather than MPICH 1.2.3.
- Comparing these packages to non-Java packages. For example, running the benchmarks using C and native threads.
- Further investigating the error messages from the *JavaSpaces* and *mpiJava* packages, and determining why there were no results for *CTJ* using four processors for the SOR benchmark.

8 Conclusions

In this study, we evaluated the ease of installation, ease of use, and performance of Java packages found that might support HPC. Three benchmark programs from *The Java Grande Benchmark Suite* [37], representing a mix of typical parallel applications, were used to evaluate this performance. All benchmark tests were run on a common SMP test platform.

Generally, packages that assumed or utilized the shared memory properties of the test system performed better than those that did not. Otherwise, the results from the benchmark tests were as expected: (a) Computation intensive benchmarks showed performance closer to the ideal than those that were communication intensive. (b) Performance results from tests with larger data sets were better than those with smaller data sets.

As a result of this study, we can recommend *JOMP* for ease of installation, ease of use, and performance. It is, however, strictly a shared memory package and thus cannot be used on a cluster or any other distributed memory system. *JCSP* is the next easiest to use. *JCSP* and *JOMP* were the only packages for which we experienced no difficulty during the entire evaluation process. Both had performance results that were close to the ideal.

In contrast, *JavaSpaces* is the most complicated to start and had the worst performance. To be fair, it was not really designed to support HPC. Programming under *JavaSpaces* requires adequate knowledge about its architecture and paradigm, and Jini must be started up before it can be used, not a simple task.

Although easy to install, *CTJ*'s special process scheduling management and real-time kernel are not friendly to the inexperienced programmers. *CTJ* does not automatically take advantage of a multiprocessor environment. Furthermore, the current communication mechanisms do not work well for HPC, but to be fair, that was not its design goal. The underlying communication mechanisms for both *CTJ* and *JavaSpaces* must be improved to be competitive in the HPC arena.

Further study is needed to determine package scalability and to compare performance with that of more traditional languages.

References

[1] C. Pancake and C. Lengauer. High-Performance Java. *Communications of The ACM*. October 2001.
[2] The Java Grande Forum Home Page. http://www.javagrande.org/. 2002.
[3] M. Allen and B. Wilkinson. *Parallel Programming*. 1999.
[4] The CSP Home Page. http://wotug.ukc.ac.uk/csp.shtml. March 2001.
[5] The MPI Home Page. http://www-unix.mcs.anl.gov/mpi/index.html.
[6] PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/pvm_home.html. April 2002.
[7] The OpenMP Home Page. http://www.openmp.org/. April 2002.
[8] Yale Linda Group. http://www.cs.yale.edu/Linda/linda.html. 2002.
[9] CSP for Java Home Page. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/explain.html.
[10] The PVM system. http://www.netlib.org/pvm3/book/node17.html.
[11] Netlib Repository at UTK and ORNL. http://www.netlib.org.
[12] The Linda System. http://www.netlib.org/pvm3/book/node16.html.
[13] G. Hilderink. Communicating Threads for Java. http://www.rt.el.utwente.nl/javapp. 2000.
[14] P. Welch. The Communication Sequential Processes for Java (JCSP) Home Page. http://www.cs.ukc.ac.uk/projects/ofa/jcsp. 2002.
[15] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. *Communicating Java Threads*. http://www.rt.el.utwente.nl/javapp/cjt/CJT-paper.PDF.
[16] G. Hilderink, J. Broenink, A. Bakkers, and N. Schaller. *Communicating Threads for Java*. 2000.
[17] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers. *Communicating Java Threads*. 1997.
[18] The *JavaMPI* homepage. http://perun.hscs.wmin.ac.uk/CSPE/software.html. 2000.
[19] The *mpiJava* Home Page. http://www.npac.syr.edu/projects/pcrc/HPJava/*mpiJava*.html. January 2000.
[20] S. Mintchev. Writing Programs in *JavaMPI*. 1997.
[21] The MPICH Home Page. http://www-unix.mcs.anl.gov/mpi/mpich/. 2002.
[22] Sun Microsystems – HPTC Home Page. http://www.sun.com/solutions/hpc/index.html. 2002.
[23] The WMPI Home Page. http://www.criticalsoftware.com/wmpi/. September 2001.
[24] Northeast Parallel Architectures Center at Syracuse University. http://www.npac.syr.edu/.
[25] The HPJava Project Home Page. http://www.npac.syr.edu/projects/pcrc/HPJava/index.html. January 2000.
[26] The JavaPVM Home Page. http://www.chmsr.gatech.edu/*jPVM*/. 1998.
[27] The *JPVM* Home Page. http://www.cs.virginia.edu/jpvm.html. 1999.
[28] The JOMP Home Page. http://www.epcc.ed.ac.uk/research/jomp/index_1.html. 2001.
[29] J. Bull, M. Westhead, M. Kambites, J. Obdrzalek. *Towards OpenMP for Java*. 2000.
[30] J. Obdrzalek, M. Bull. *JOMP Application Program Interface*. August 2000.
[31] D. Rossi. The Jada Home Page. http://www.cs.unibo.it/~rossi/jada/. 2001.
[32] Sun Microsystems. The JavaSpace Techology Home Page. http://java.sun.com/products/javaspaces. 2001.
[33] D. Rossi and P. Ciancarini. *Jada: A Coordination Toolkit for Java*. 1997.
[34] Sun Microsystems. *JavaSpaces™ Service Specification*. December 2001.
[35] Jini Network Technology Developer Center for Sun Microsystems, Inc. http://developer.java.sun.com/developer/products/jini/. April 2002.
[36] Quickstone Technologies. http://www.quickstone.com/.
[37] The JavaG Benchmarking Home Page. http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html. June 2001.
[38] L. Smith, J. Bull, and J. Obdrzalek. *A Parallel Java Grande Benchmark Suite*. November 2001

Appendix A – The Raw Data for Figures 1 – 6

	Execution Time (Sec.)				Speedup			
Package\Proc.	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
CTJ	347.46	188.25	137.96	116.34	0.96	1.78	2.43	2.88
Jada	334.58	168.25	112.49	85.37	1.00	1.99	2.97	3.92
JavaSpaces	*	*	*	*	*	*	*	*
JCSP	333.66	168.29	112.68	85.74	1.00	1.99	2.97	3.90
JOMP	332.14	168.80	115.60	86.62	1.01	1.98	2.89	3.86
mpiJava	340.93	177.04	*	94.20	0.98	1.89	*	3.55
Sequential	334.66							

Table 3. Crypt Benchmark Data

	Execution Time (Sec.)				Speedup			
Package\Proc.	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
CTJ	709.75	355.43	243.45	196.58	0.99	1.97	2.88	3.57
Jada	701.07	353.48	242.34	198.22	1.00	1.99	2.90	3.54
JavaSpaces	724.96	371.73	245.32	194.26	0.97	1.89	2.86	3.61
JCSP	686.00	345.58	238.72	178.51	1.02	2.03	2.94	3.93
JOMP	714.54	357.47	246.02	179.69	0.98	1.96	2.85	3.91
mpiJava	726.98	363.68	246.70	182.36	0.97	1.93	2.85	3.85
Sequential	701.95							

Table 4. Series Benchmark Data

	Execution Time (Sec.)				Speedup			
Package\Proc.	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
CTJ	144.72	119.31	114.71	113.93	0.80	0.97	1.01	1.02
Jada	139.10	71.17	48.87	45.58	0.83	1.63	2.37	2.54
JavaSpaces	154.51	113.01	98.70	106.32	0.75	1.03	1.17	1.09
JCSP	132.76	67.01	45.11	35.32	0.87	1.73	2.57	3.28
JOMP	133.58	67.91	47.07	37.31	0.87	1.71	2.46	3.11
mpiJava	138.50	*	*	*	0.84	*	*	*
Sequential	115.85							

Table 5. SOR Benchmark Data