A Parallel Java Grande Benchmark Suite

L. A. Smith and J. M. Bull
EPCC, The King's Buildings, The University of Edinburgh,
Mayfield Road, Edinburgh, EH9 3JZ, Scotland, U.K.
email: `epcc-javagrande@epcc.ed.ac.uk`

J. Obdržálek
Faculty of Informatics, Masaryk University,
Botanická 68a, 602 00 Brno, Czech Republic.
email: `xobdrzal@fi.muni.cz`

Abstract

Increasing interest is being shown in the use of Java for large scale or *Grande* applications. This new use of Java places specific demands on the Java execution environments that can be tested using the Java Grande benchmark suite [5], [6], [7]. The large processing requirements of *Grande* applications makes parallelisation of interest. A suite of parallel benchmarks has been developed from the serial Java Grande benchmark suite, using three parallel programming models: Java native threads, MPJ (a message passing interface) and JOMP (a set of OpenMP-like directives). The contents of the suite are described, and results presented for a number of platforms.

Keywords: Java, parallel, benchmarking, message passing, threads.

1 Introduction

Java offers a number of benefits as a language for High Performance Computing (HPC), especially in the context of the Computational Grid[1]. For example, Java offers a high level of platform independence not observed with traditional HPC languages. This is an advantage in an area where the lifetime of application codes exceeds that of most machines. In addition, the object-oriented nature of Java facilitates code re-use and reduces development time. However, there are a number of outstanding issues surrounding the use of Java for HPC, principally: performance, numerical concerns and lack of standardised parallel programming models.

The serial Java Grande benchmark suite provides a standard benchmark suite for computationally intensive applications. For further details of the serial benchmarks, the rationale for their design, and reported results see [5], [6] and [7]. In creating a parallel version of the suite, the aim is to provide a means of evaluating the emerging parallel programming paradigms for Java, and to expose key weaknesses which must be addressed before parallel Java applications can be viable for HPC.

There is a wide variety of interfaces and language extensions for parallel and distributed programming in Java. Both Java threads and Remote Method Invocation (RMI) are part of the Java specification. Java threads, although principally designed for concurrent, rather than parallel, programming can successfully be used on shared memory multiprocessors. RMI is not well suited to parallel programming, both due to its programming paradigm and its high overheads. Furthermore, a comprehensive suite of benchmarks developed at the University of Karlsruhe [11] is already available.

The two other interfaces which we have used in the parallel benchmark suite are MPJ [2] and JOMP [10]. These are prototype specifications of Java counterparts to MPI and OpenMP respectively. We have chosen these interfaces due to the familiarity and widespread use of their Fortran and C predecessors, and the fact that neither requires extension to the core Java language. It should be noted that neither is yet standardised, and so may be subject to change in the future.

MPJ consists of a class library providing an interface for message passing, similar to the MPI interface for C and Fortran. Most of the functionality found in MPI is supported, and messages may consist of arrays of either basic types or of objects. Existing implementations such as mpiJava [3] use the Java Native Interface (JNI) mechanism to call existing MPI libraries written in C. However, research efforts are underway to provide pure Java implementations using sockets or VIA.

JOMP is a specification of directives (embedded in standard Java as comments), runtime properties and a class library similar to the OpenMP interface for C and Fortran. The existing implementation [8], [10] uses a source-to-source translator (itself written in Java) to convert the directives to calls to a runtime library, which in turns uses the standard Java threads interface. The system is pure Java, and therefore transparently portable.

Other approaches to providing parallel programming interfaces for Java include JavaParty [12], HPJava [9], Titanium [14] and SPAR Java [13]. These are also in the research phase and, in addition, require genuine language extensions.

The remainder of this paper is structured as follows: in Section 2 we describe the structure and contents of the benchmark suite. In Section 3 we describe the systems and environments tested, and present and discuss the results obtained. Section 5 provides some conclusions.

2 The Parallel Benchmark Suite

2.1 Section I

Section I of the serial Java Grande suite consists of microbenchmarks for low-level operations such as arithmetic, maths library functions, object creation, exception handling and serialisation. Garbage collection is also an important feature of Java which can have a significant impact on performance, though this may not be so relevant to scientific applications which tend to have long-lived data structures. We have not included a low-level garbage collection benchmark, as it is very difficult to devise such a benchmark which is either representative of allocation/deallocation patterns in real applications, or is robust against optimisation. For kernels and applications, it is not possible to determine the time spent doing garbage collection without instrumenting the garbage collector itself. This we are unable to do within the context of the benchmark suite.

Parallel versions of the existing low-level operations are of little interest, so in the parallel suite we have developed new low-level benchmarks which are pertinent to each of the the three parallel programming models.

2.1.1 Threads

Barrier Benchmark This measures the performance of barrier synchronisation. Two types of barriers have been implemented: the Simple Barrier uses a shared counter, while the Tournament Barrier uses a lock-free 4-ary tree algorithm.

ForkJoin Benchmark This benchmark measures the time spent creating and joining threads.

Synchronisation Benchmark This benchmark measures the performance of synchronized methods and synchronized blocks under contention.

2.1.2 JOMP

Synchronisation This benchmark measures the overhead of synchronisation directives such as `parallel`, `for`, `parallelfor`, `barrier`, `critical` and `single`. The method used is identical to the OpenMP microbenchmark suite described in [4].

Scheduling This benchmark measures the overhead of the various loop scheduling options (static, dynamic and guided) with varying chunk sizes.

2.1.3 MPJ

Barrier This measures the cost of barrier synchronisation.

PingPong This measures the costs of point-to-point communication for a range of message lengths. Two different data types are considered: a basic type (double) and an object (containing a double as a data member).

Bcast, Alltoall, Gather, Scatter, Reduce These benchmarks measure the cost of the collective communications Broadcast, All-to-all, Gather, Scatter and Reduce. Where possible, both basic and object types are considered.

2.2 Section II

Section II of the parallel benchmark suite contains parallel versions of a subset of Section II of the serial suite. These codes are simple kernels which reflect the type of computation which might be found in the most computationally intense parts of real numerical application. While some of these kernels are "embarrassingly parallel", they nonetheless provide valuable information about the scalability of the parallel Java environment.

Series Benchmark

This 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. Each iteration of the loop is independent of every other loop and the work may be distributed simply between processors.

SOR Benchmark

The serial version of this benchmark performs 100 iterations of successive over-relaxation on a $N \times N$ grid. The performance reported is in iterations per second. This benchmark involves an outer loop over iterations and two inner loops, each looping over the grid. In order to update elements of the principle array during each iteration, neighbouring elements of the array are required, including elements previously updated. Hence this benchmark is, in this form, inherently serial. To allow parallelisation 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 in a block manner.

LUFact Benchmark

This benchmark solves an $N \times N$ linear system using LU factorisation followed by a triangular solve. This is a Java version of the well known Linpack benchmark. The factorisation is the only part of the computation performed which is parallelised: the remainder is computed in serial. Iterations of the double loop over the trailing block of the matrix are independent and the work is divided between the processors in a cyclic fashion.

Crypt Benchmark

Crypt performs IDEA (International Data Encryption Algorithm) encryption and decryption of an array of N bytes. This algorithm involves two principle loops, whose iterations are independent and are divided between processors in a block fashion.

SparseMatMult Benchmark

This uses an unstructured sparse matrix stored in compressed row format with a prescribed sparsity structure. An $N \times N$ matrix is used for 200 iterations. The principle computation involves an outer loop over iterations and an inner loop over the size of the principal arrays. The simplest parallelisation mechanism is to divide the loop over the array length between processors. Parallelising this loop creates the potential for more than one thread to up-date the same element of the result vector. To avoid this the non zero elements are sorted by their row value. The loop has then been parallelised by dividing the iterations into blocks, which are approximately equal, but adjusted to ensure that no row is access by more than one thread.

2.3 Section III

Section III of the parallel benchmark suite contains parallel versions of a subset of Section III of the serial suite. These are larger codes intended to be representative of real numerical applications.

MolDyn Benchmark

This benchmark is an $O((N * (N-1))/2)$ N-body code modelling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. The computationally intense component of the benchmark is the force calculation, which calculates the force on a particle in a pair wise manner. This involves an outer loop over all particles in the system and an inner loop ranging from the current particle number to the total number of particles. The outer loop has been parallelised by dividing the range of the iterations of the outer loop between processors, in a cyclic manner to avoid load imbalance.

MonteCarlo Benchmark

This benchmark is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean fluctuation as a series of historical data. The principle loop over number of Monte Carlo runs can be easily parallelised by dividing the work in a block like fashion.

RayTracer Benchmark

This benchmark measures the performance of a 3D ray tracer. The scene contains 64 spheres and is rendered at a resolution of $N \times N$ pixels. The outermost loop (over rows of pixels) has been parallelised using a cyclic distribution for load balance.

3 Results

Running the entire benchmark suite produces a large quantity of data, so in this section we present a selection to illustrate the types of results which can be obtained. The benchmark suite was run on a Sun HPC 6500 system with 18 400MHz UltraSparc II processors and 18 Gbytes of memory running Solaris 2.7, and a SGI Origin 3000 system[1] with 128 400 MHz MIPS R12000 processors and 128 Gbytes of main memory running IRIX 6.5.

The execution environments (all 32-bit) utilised were Sun JDK 1.2.1_04, Sun JDK 1.3.0 and Sun JDK 1.3.1 on the HPC 6500 and SGI JDK 1.3.0 on the Origin 3000. The three Sun JDKs suffer from a performance bug, which under some circumstances results in the number of Solaris Lightweight Processes (LWPs) being smaller than the number of Java threads. This causes the threads to share CPU resources, resulting is a significant loss of scalability. We have attempted to avoid the effects of this bug by using the JNI to make a call to the Solaris threads library routine `thr_setconcurrency()`.

3.1 Section I

3.1.1 Threads

Figure 1 shows the performance of the Section I benchmarks Barrier and ForkJoin on the three Sun JDKs. This figure clearly demonstrates the need for using fast barrier synchronisation and avoiding thread creation in shared memory programs. Thread fork/join requires several milliseconds, which is unacceptable for anything but the most coarse-grained problems. The simple barrier (using wait/notify) is faster than fork/join, but still two orders of magnitude slower than the tournament barrier. Of the three Sun JDKs, the performance of the 1.2.1 version (which uses the Classic VM) is slightly better than the 1.3.1 version (which uses the HotSpot VM), and clearly superior to that of the 1.3.0 version (which also uses the HotSpot VM). In all the results presented here the 1.3 JDKs were run with the `-server` option. We also tested the `-client` option but observed only minor differences.

3.1.2 MPJ

Figure 2 shows the performance of the Section I benchmarks Bcast and PingPong with increasing array size. Results are presented on two processes for the PingPong benchmark and 16 processes for the Bcast benchmark. These results clearly demonstrate that considerable performance loss is encountered when sending an array of objects (each object containing a double) over sending an array of doubles. This overhead is due to serialisation of the objects, which introduces a large software overhead—reducing serialisation overheads is an area of significant current research by other groups. Little performance variation exists between the three Sun JDKs.

[1]We greatfully acknowledge CSAR for early access to this machine.

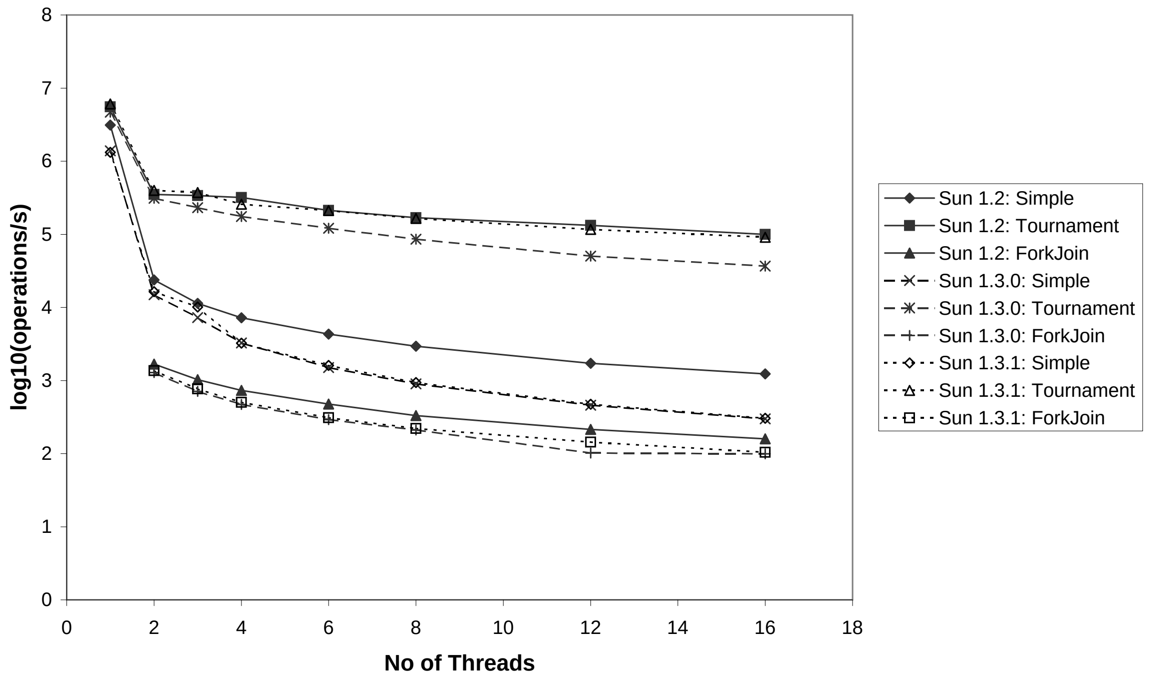

Figure 1: Performance of the low-level Threaded benchmarks Barrier and ForkJoin.

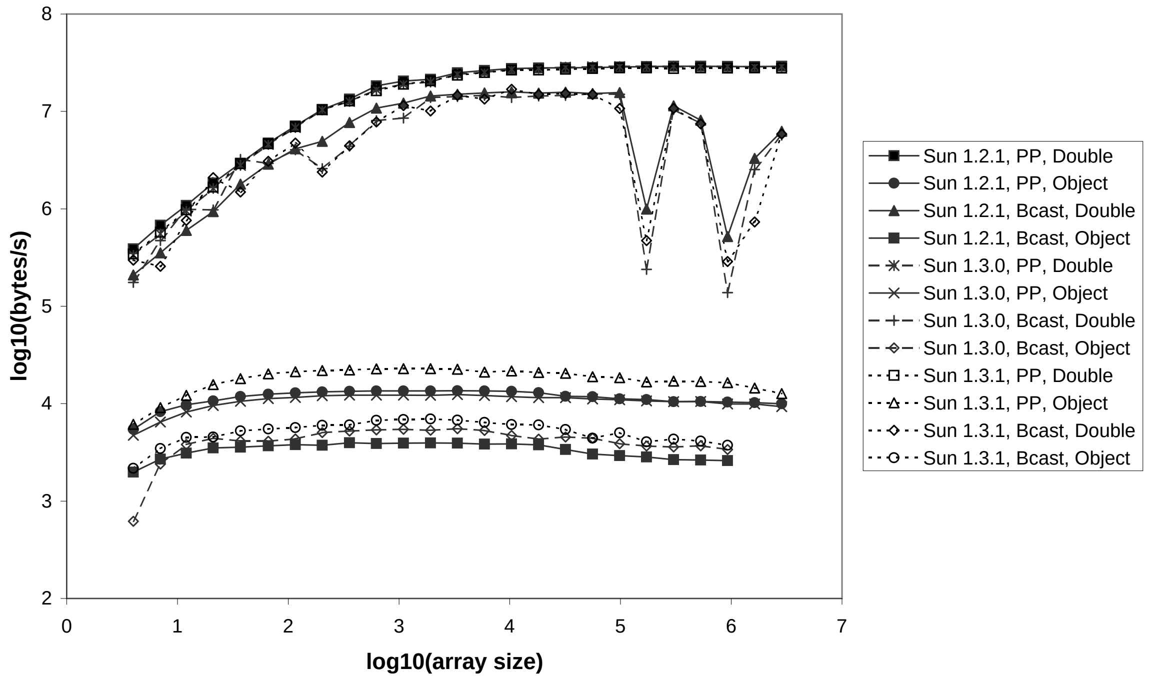

Figure 2: Performance of the low-level MPJ benchmarks Bcast and PingPong.

3.1.3 JOMP

Figure 3 shows the overhead in clock cycles measured for the `parallel`, `for`, `barrier` and `critical` directives by the Section I Synchronisation benchmark. Examining the three Sun JDKs, the performance of the 1.2.1 version is superior to that of both the 1.3 versions for the directives `parallel`, and `for`, and superior to the 1.3.0 version for the `barrier` directive. The performance of the 1.3.0 version on the `parallel` and `for` directives and the 1.3.1 version on the `for` directive is particularly poor, showing considerable overhead increase between 12 and 16 processors. Examining the `critical` results shows the 1.3 versions have better performance at lower thread numbers, however the 1.2.1 version scales better, and has lower overhead with 16 threads.

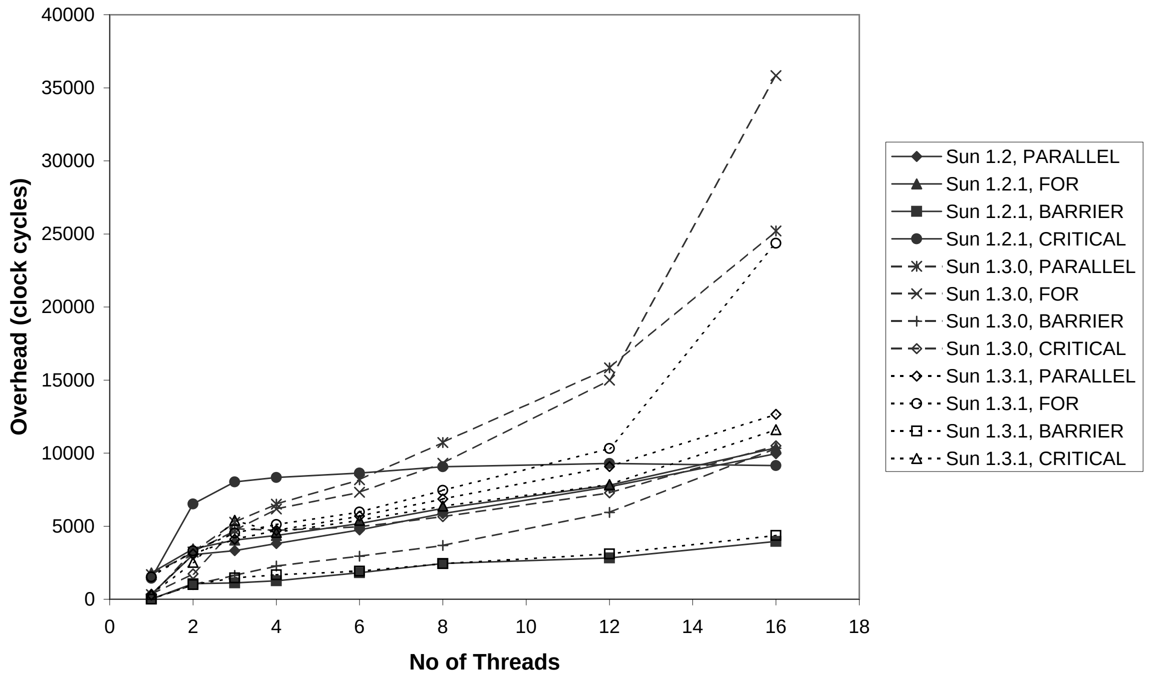

Figure 3: Performance of the low-level JOMP directives `parallel`, `for`, `barrier` and `critical`.

3.2 Section II

Figure 4 shows the speedup of the Crypt benchmark from Section II of the suite. The results presented correspond to an array of $50,000,000$ bytes. The serial execution times of the threaded, JOMP and MPJ benchmarks are similar for each JDK. Variation exists between JDKs however, with a slower execution time for JDK 1.3.0 over JDK 1.2.1 and JDK 1.3.1. This algorithm is trivially parallel, involving two principle loops whose iterations are independent and can be divided between processors in a block fashion. The speed-up of the threaded and JOMP implementations is therefore reasonable, with all showing almost ideal speed-up. The MPJ implementation however shows poorer results. Whilst parallelisation is still obtained over the two principle loop iterations, the MPJ code involves the communication of sub-array data at the end of the computation, creating a communication overhead.

Figure 5 shows the speedup of the LUFact benchmark from Section II of the suite. The results presented correspond to matrix dimensions of 2000×2000. Despite similar serial execution times, the Sun 1.3.0 JVM shows slightly poorer speedup than the 1.2.1 version, reflecting the higher cost of barrier synchronisation in the 1.3.0 JVM observed in the Section I benchmark. The Sun 1.3.1 JVM shows better speed-up than any of the other JVMs. In addition, an overall improvement in performance is observed, with execution times typically around two thirds of the older JVMs. The performance of Java threads and JOMP are very similar, and a great deal superior to the MPJ results. Whilst parallelisation is achieved in a similar manner for each of the three benchmarks, the MPJ code contains explicit communication for each iteration of the outer loop (over columns of the matrix). A copy of the pivot index and pivot column is broadcast to every process creating an overhead not present within the threaded and JOMP implementations. Results are also presented for the SGI, clearly demonstrating poorer performance than the SUN JDKs.

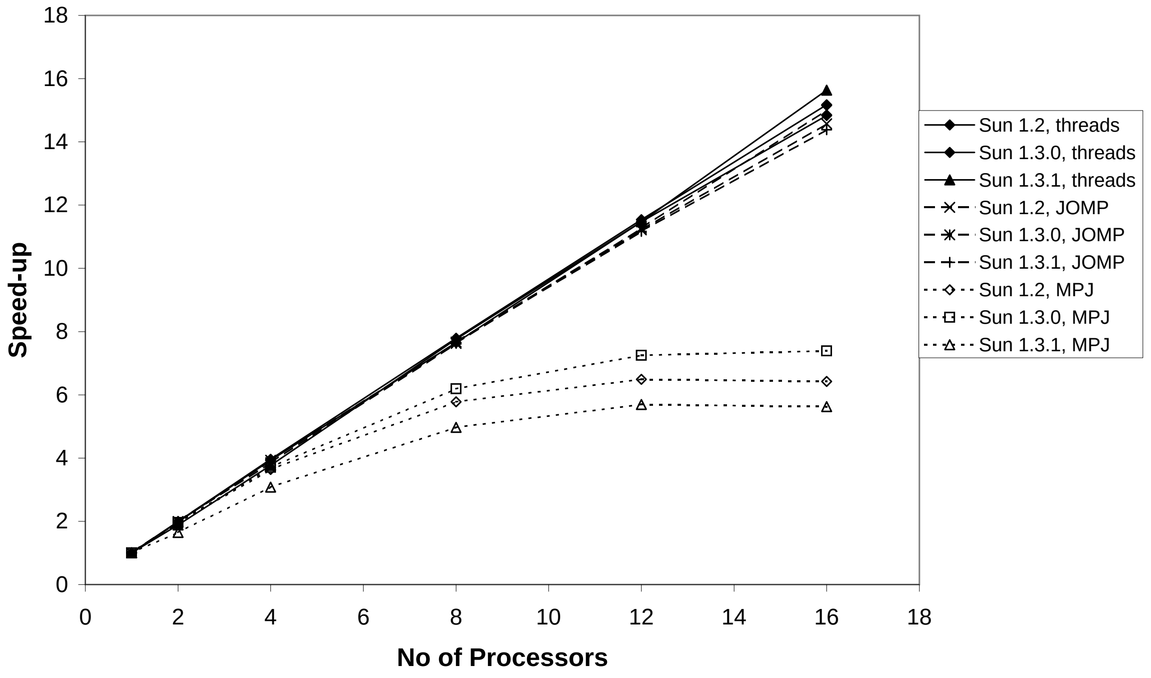

Figure 4: Performance of the Section II Crypt benchmark.

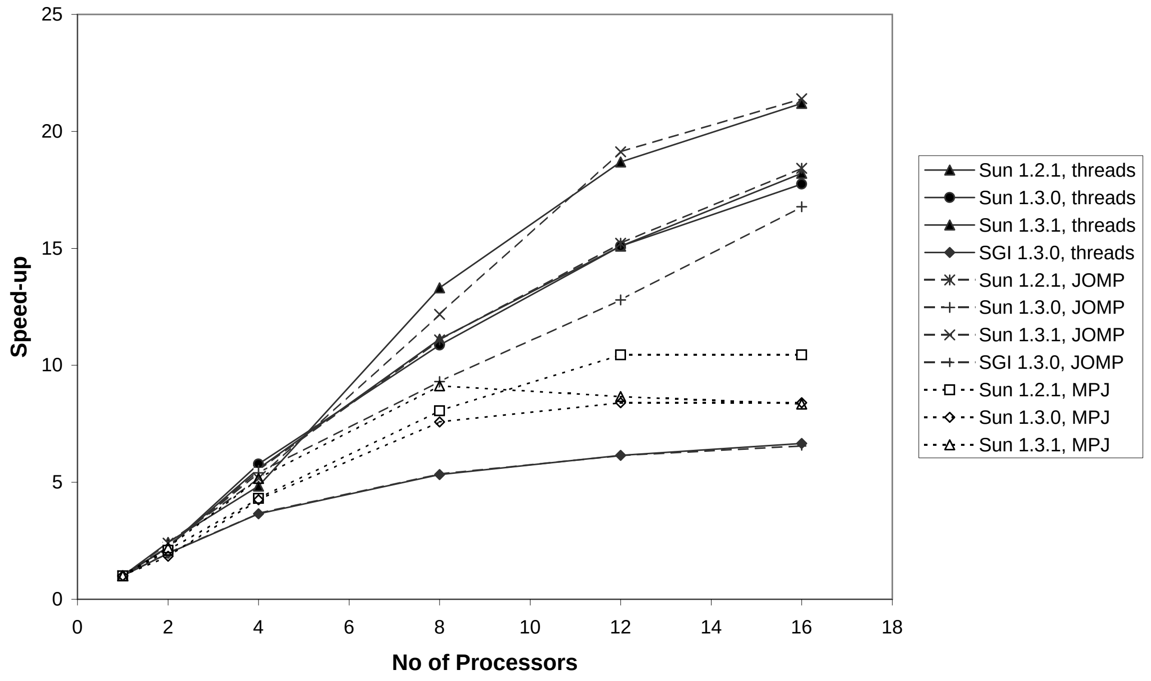

Figure 5: Performance of the Section II LU Factorisation benchmark

4 Section III

Figure 6 shows the speedup of the RayTracer benchmark from Section III, using data size B (500×500 pixels). As with the previous example, the Sun 1.3.0 JVM shows poorer speedup than the 1.2.1 version. The Sun 1.3.1 JVM shows slightly better speedup than the 1.2.1 version on the threaded benchmark, but poorer performance elsewhere. The SGI results demonstrate poorer performance than all the SUN JDKs. Unlike the previous example however, the MPJ results are considerably better than the threaded and JOMP benchmarks with the JOMP benchmarks demonstrating the poorest performance. The good performance of the MPJ benchmark is due to the lack of communication required within the benchmark, with only the cost of returning partial copies of the pixel array to the master process required at the end of the parallel computation. The poor performance of the JOMP benchmark is a result of the presence of a global shared copy of the scene and environment. Both the threaded and MPJ benchmarks have private copies of these, allowing sequential optimisations, such as the use of class variables for temporary storage, to be carried over to the parallel code.

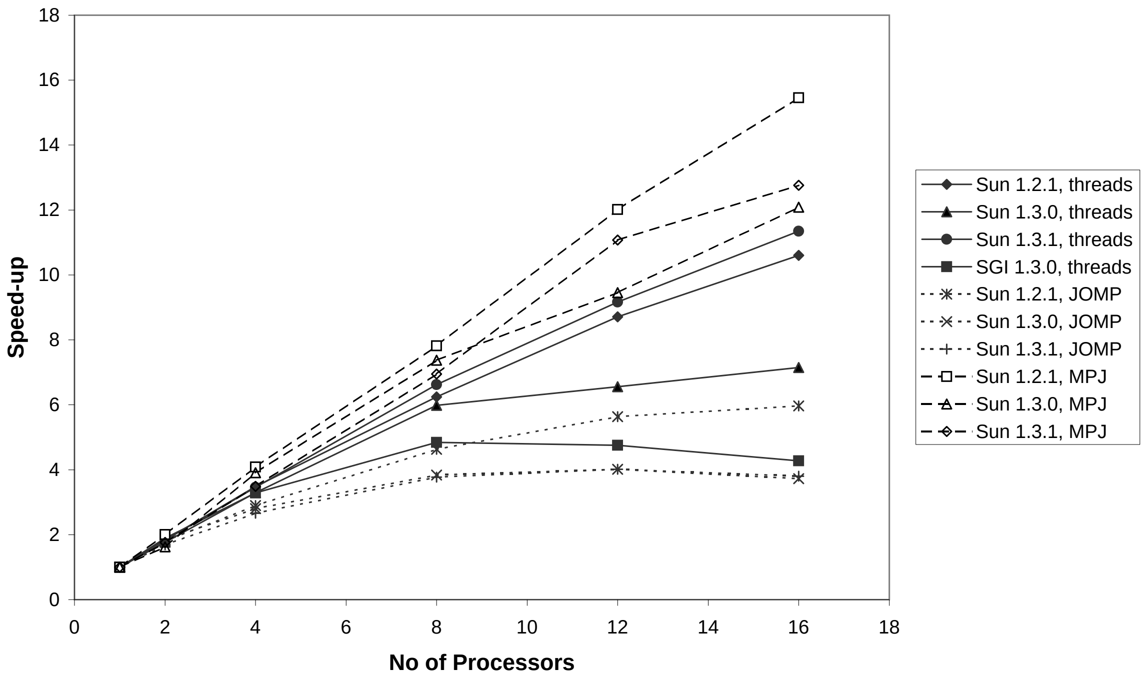

Figure 6: Performance of the Section III RayTracer benchmark.

Figure 7 shows the speedup of the Monte Carlo benchmark from Section III, using data size B ($60,000$ sample time series). This benchmark exhibits poor speedup on all JDKs, possibly as a result of memory management causing a bottleneck. In this case, the Sun 1.3.0 JDK demonstrates greater speedup than the Sun 1.3.1 and 1.2.1 JDKs. This may be due to more efficient memory allocation and garbage collection by the Sun JDK 1.3.0. In addition some performance is lost due to the results being stored in a Vector array, with the Synchronized addElement method used to add objects to the array. The MPJ benchmarks exhibit poorer speedup than the threaded and JOMP benchmarks, a result of returning partial copies of the results array to the master process and the sequential addition of these objects to the Vector results array.

5 Conclusions and future work

We have presented a parallel Java Grande benchmark suite which can be used to test parallel execution on shared and distributed memory architectures. The results shown here demonstrate some interesting differences between programming models, Java execution environments, and hardware platforms. They show that although parallel programming in Java can exhibit reasonable scalability, the software support is still relatively immature. Future work will focus on examining a wider range of platform and Java environments, and extending the benchmark suite to include codes which use more complex parallel algorithms.

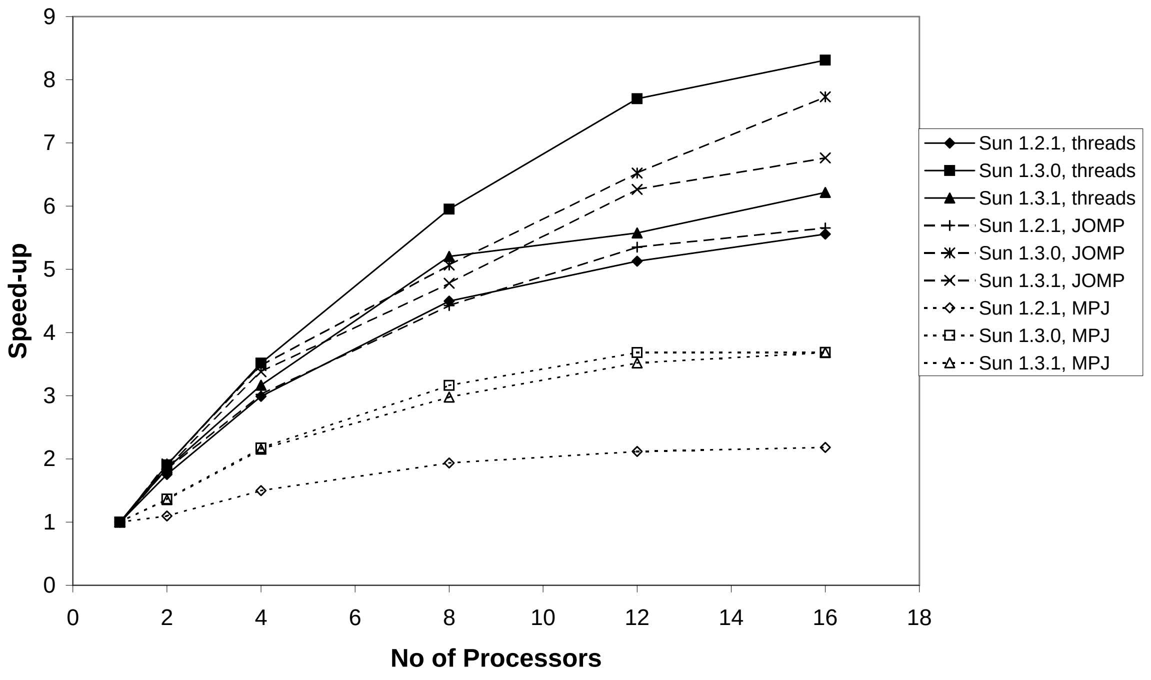

Figure 7: Performance of the Section III Monte Carlo benchmark

References

[1] Foster I. and Kesselman, C. (1999) *The Grid: Blueprint for a New Computing Infrastructure*, published by Morgan Kaufmann.

[2] Baker, M.A. and Carpenter, D.B. (2000) *MPJ: A Proposed Java Message-Passing API and Environment for High Performance Computing*, in Proceedings of Second Java Workshop at IPDPS 2000, Cancun, Mexico, LNCS, Springer Verlag, Heidelberg, Germany, pp. 552-559.

[3] Baker, M., Carpenter, B., Fox, G., Ko, S.H., and Lim S. (1999) *mpiJava: An Object-Oriented Java interface to MPI*, in Proc. International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, April 1999.

[4] Bull, J.M. (2000) *Measuring Synchronisation and Scheduling Overheads in OpenMP*, in Proceedings of First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pp. 99–105.

[5] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A. (2000) *A Benchmark Suite for High Performance Java*, Concurrency, Practice and Experience, vol. 12, pp. 375-388.

[6] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A. (2000) *Benchmarking Java Grande Applications*, in Proceedings of the Second International Conference on The Practical Applications of Java, Manchester, U.K., April 2000, pp. 63-73.

[7] Bull, J.M., Smith, L.A., Westhead, M.D., Henty, D.S. and Davey, R.A. (1999) *A Methodology for Benchmarking Java Grande Applications*, in Proceedings of ACM 1999 Java Grande Conference, June 1999, ACM Press, pp. 81-88.

[8] Bull, J.M. and Kambites, M. E. (2000) *JOMP—an OpenMP-like Interface for Java*, in Proceedings of the ACM 2000 Java Grande Conference, June 2000, pp. 44-53.

[9] Carpenter, B., Zhang, G., Fox, G., Li, X., and Wen, Y. (1998) *HPJava: Data Parallel Extensions to Java*, Concurrency: Practice and Experience, vol. 10, pp. 873-877.

[10] Kambites, M.E., Obdrzalek, J. and Bull, J.M. (2001) *An OpenMP-like Interface for Parallel Programming in Java*, to appear in Concurrency and Computation: Practice and Experience.

[11] Nester, C., Philippsen, M., Haumacher, B., (1999) *A more efficient RMI for Java*, Proceedings of ACM 1999 Java Grande Conference, June 1999, ACM Press, pp. 152-159.

[12] Philippsen, M. and Zenge, M. (1997) *JavaParty—Transparent Remote Objects in Java*, in Concurrency: Practice and Experience, vol. 9, pp. 1225-1242.

[13] van Reeuwijk, K., van Gemund, A.J.C. and Sips, H.J. (1997) *SPAR: A Programming Language for Semi-automatic Compilat ion of Parallel Programs*, in Concurrency: Practice and Experience, vol. 9, pp. 1193-1205.

[14] Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P. and Aiken, A. (1998) *Titanium: A High-Performance Java Dialect*, in Concurrency: Practice and Experience, vol. 10, pp. 825-836.