ELSEVIER

Advances in Engineering Software 31 (2000) 687–696

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

A test suite for high-performance parallel Java

J. Häuser[a], T. Ludewig[a], R.D. Williams[b], R. Winkelmann[a], T. Gollnick[a], S. Brunett[b], J. Muylaert[c]

[a]Department of High Performance Computing, Center of Logistics and Expertsystems GmbH, Salzgitter, Germany
[b]Center for Advanced Computing Research, California Institute of Technology, Pasadena, CA, USA
[c]European Space Agency, ESTEC, Noordwijk, Netherlands

Abstract

The Java programming language has a number of features that make it attractive for writing high-quality, portable parallel programs. A pure object formulation, strong typing and the exception model make programs easier to create, debug and maintain. The elegant threading provides a simple route to parallelism on shared-memory machines. Anticipating great improvements in numerical performance, this paper presents a suite of simple programs that indicate how a pure Java Navier–Stokes solver might perform. The suite includes a parallel Euler solver. We present results from a 32-processor Hewlett–Packard machine and a 4-processor Sun server. While speedup is excellent on both machines, indicating a high-quality thread scheduler, the single-processor performance needs much improvement. © 2000 Published by Elsevier Science Ltd. All rights reserved.

Keywords: Java; Parallel programs; Navier–Stokes

1. A high-performance Java test suite for engineering and science

In this paper we present a set of parallel Java [1,2] programs based on the Java Thread concept as well as the Java Remote Method Invocation (RMI) to serve as a test suite to measure the suitability of the Java OOP [3,4] (Object-Oriented-Programming) methodology to be used for large scale applications in engineering and science. We will provide quantitative measurements concerning parallel speedup and parallel scalability as well as *thread* handling in Java. Also, the question of numerical performance will be addressed and the impact of the various Java compilers will be presented.

In addition, we will also discuss qualitative questions such as code robustness, maintainability, reusability and testability. In light of the frustrating failures of launches by NASA, Lockheed Martin or Boeing that resulted in multibillion-dollar losses the question of software reliability is a major topic. Also, in Europe, flight 501 ended in self-destruction of the new launcher Ariane 5 after 37 s in flight, because of a programming error.

We will first discuss why Java is the language of choice for high-performance applications in science and engineering. We will present computational evidence as well as code fragments in order to demonstrate that Java can provide both requirements, namely high numerical performance on parallel architectures and all the conceptual benefits of OOP. Java allows the code developer to think in systems and not in algorithms. Thus, the code design process reflects the actual engineering system and therefore displays a much clearer design and visibility.

While the impact of the Java programming language has not yet been strongly felt in the high-performance computing community, it has quickly become the de facto language in the huge web programming industry. We believe that this trend will continue into numerically intensive computing [5,6] because a Java program is more flexible, easier to debug and easier to write than a program in C++ or Fortran. Furthermore it has convenient and flexible threading to allow the programmer to make parallel programs.

When we compared Java with existing programming languages for science and engineering, namely Fortran and C (to some extent C++), despite its similar syntax to C, it became clear that Java was not just another programming language. Java is an OOP language, providing, however, a much cleaner design than C++. OOP allows code construction reflecting, for instance, the engineering design process, because objects can be software coded and integrated. In addition, Java is the programming language for the Internet, and thus Java objects on disparate machines or even separate networks can be connected.

Producing engineering software in Java requires a different way of thinking, i.e. central to Java is the class concept. A class is a collection of data structures and methods, describing the functionality of a certain item, for example, a wing. An aircraft can be described by a set of classes, representing a wing, fuselage, nacelle, pylon, engine, etc.

0965-9978/00/$ - see front matter © 2000 Published by Elsevier Science Ltd. All rights reserved.
PII: S0965-9978(00)00029-6

ClibPDF - www.fastio.com

A specific aircraft can be constructed by instantiating objects from these classes. In this way, a direct mapping of the engineering parts to the corresponding software objects can be achieved. The Java language mechanism allows encapsulation and inheritance, meaning that an existing class can be used and modified according to the needs of the code designer. The validated parent class will not be touched, allowing complete code reusability. The interface notion of Java extends the concept of inheritance, providing some kind of template. The Java OOP approach provides profoundly improved software productivity. Java's robust mechanism for exception handling promotes code reliability (remember the Ariane 5 flight 501 in 1996 when the launcher was destroyed by the self destruction system because of software error, or faulty software involved in the recent failures that occurred in six of the last nine U.S. launches), a feature considered to be essential for today's large and complex codes.

Besides the clean, OOP model, Java makes it much easier for a programmer to use *threads*, which is the primary route to parallelism (and therefore speed) on a shared-memory machine—the architecture of many modern machines.

Java has a unique capability of using the Internet. This special feature has been used to build a general client–server application that allows the client and the server to communicate and exchange data as well as code over the Internet. The idea is to provide a general framework for simulation purposes governed by partial differential equations where at run time the client proves the specific numerical solver to the server. Once this framework is in place, it can serve as a parallel and geometric platform for any kind of scientific or engineering problem described by a solution space and a set of equations.

1.1. Why threads are good for CFD

When writing a parallel program, the memory of the machine may be distributed or shared. In the former case, each processor has its own memory, and communication is by messages; in the latter case, each processor has access to all of the memory.

Distributed memory is easier and therefore cheaper to implement, but it leaves the programmer responsible for partitioning the data among the processors. For a simple CFD solver this approach is very effective: the computational grid is partitioned and a messaging scheme is coded to exchange data between neighboring processors that share a boundary. After each time step, there is a loose synchronization mediated by this data exchange, so the simulation moves at the pace of the slowest processor. Therefore, we require each processor to have roughly equal work in order to obtain good performance from the machine. If each grid point takes equal work to update it, then we can split the grid so that each processor has the same number of grid points. This strategy works very well for simple, explicit schemes [7].

Unfortunately, the most numerically efficient CFD schemes do not generate the same computational work at each grid point. An iterative linear solver, employed separately on each block of a multiblock grid, may require different numbers of iterations on each of the blocks. Grid points may differ in the algorithms they use: different temperatures or shear rates may cause different turbulence or chemistry models to be invoked. Furthermore there may be different subgrid schemes near the boundaries. All of these effects point to increasing computational heterogeneity with increasing sophistication of the solver.

Complex dynamic load-balancing schemes have been proposed and implemented on distributed-memory machines [8]. However, we feel that it is worth considering a different approach, where the processors are kept busy by self-scheduling and shared memory. The computational load is divided into a set of threads (many more threads than processors), and when a processor becomes idle, it goes to the thread pool, removes a thread and works on it, then puts it back in the pool and gets another. In the JParEuler solver discussed below, a thread is associated with a section of a CFD grid. Each thread needs information from other threads (neighbors in the grid) before it can continue: the programming model has constructs that make sure this is available before a processor starts to work on the thread. Once a time-step has been executed, the thread goes back in the pool until its neighbors have been updated to the same level, and then it can be updated another time-step.

Thus, the burden of parallel computing is not borne solely by the CFD implementer, but is shared with the operating system design team; the quality of thread scheduling and memory subsystem directly influence the efficiency of the code. Moreover, this approach directly provides dynamic load balancing.

1.2. Java numerical performance

One of the attractive features of Java is its portability, the idea of "write once, run anywhere". While this is not entirely true, the experience of these authors is that porting Java is certainly easier than porting C, C++ or Fortran codes. The portability is obtained because Java is generally not compiled to a platform-specific executable, but converted to so-called *byte code*, which is in turn executed by a platform-specific interpreter. While these extra layers of insulation provide portability, they also have a serious performance impact. However, many companies, including Hewlett–Packard, IBM [9,10] and Sun, offer (or will offer) Java compilers that create native code directly, and should provide competitive performance with C or Fortran. Thus a user has the choice of portability or performance; the user can develop with one model, then run with the other.

Another reason why Java is slower than C is that a garbage-collector thread is always running; reclaiming memory that is no longer needed by the application. It is

ClibPDF - www.fastio.com

this overhead which takes away resources from the numerical application, but at the same time it provides an enormous boost to programmer productivity. The programmer is thinking about CFD code instead of spending hours chasing mysterious memory leaks that *always* beset large C++ projects.

Multithreading is a way to get improved performance from a code, because many machines have extra processors that can run the extra threads. But in C it is much more difficult to manage threads than it is in Java; therefore, programmers simply do not use threads very much. But in Java, it is very easy to spawn a new thread. Therefore, use of threads is much more natural and widespread.

Although Java programs are statically compiled, there is still a need to do some runtime checking. In particular, null references checking, array bounds checking and runtime type checking cannot be done at compile time. This makes Java programs more robust, but it also makes the generated code a little slower than the equivalent C program. However, many of these checks can be eliminated at runtime by the native code generator.

1.3. JParNSS

We are in the process of producing a parallel Navier–Stokes solver ***JParNSS*** [11,12], using the principles learned from this test suite. The test suite presented here thus concentrates on two aspects of the performance: the single-node performance of the Java interpreter; and the speedup (ratio of single-node speed to multiple-node speed) that is obtained. We have run the test suite on two machines at Caltech, one a super computer with 32 nodes from Hewlett–Packard, the other a 4-processor Sun machine running Solaris.

2. Test machines

2.1. HP V-class

In August 1999 the Hewlett–Packard V-Class server at the Center of Advanced Computing Research, California Institute of Technology had 32 PA-8200 processors running at 240 MHz, with 16 GB RAM. Each processor has a 2 MB instruction as well as data cache. Within a node the memory access is implemented by the HyperPlane crossbar technology that provides high bandwidth and low latency access from CPU as well as I/O to local memory. This nonblocking 8×8 crossbar provides a maximum of 15.36 GB/s memory bandwidth with I-directional 960 MB/s per port. Communication across nodes is achieved by HyperLinks providing a peak bandwidth of 3.84 GB/s. The operating system is HP-UX version 11.01. HP provided their latest Java version (Java 1.1.7) that is running on the HP V-Class.

2.2. Sun Enterprise E450

The Sun Enterprise 450 has an SMP architecture, utilizing four 300 MHz Ultra SPARC II processors that are connected by a 1.6 GB/s UPA interconnected to 1.7 GB shared memory. Operating system version 5.6 in combination with Java 1.2 (Solaris VM (build Solaris_JDK_1.2_01_dev06_fcsV) native threads sunwjit) was used on this machine.

2.3. Pentium II 300 MHz, Linux

For comparison we also used a 300 MHz Pentium II PC under Linux along with the IBM Alpha Works Java compiler 1.1.6 that was available as a prerelease version without optimization. We found that this compiler produced fast code, and we expect future releases to rival the speed of highly optimized C code.

3. The test suite

3.1. Square root

In this ultra-simple program, many identical threads are used for simple arithmetic—computing square roots. It is an embarrassingly parallel problem, meaning that the threads do not have to communicate, and thus there is no need for thread synchronization. This is rather the exception than the rule. In general, threads need to communicate as will be shown in the Laplace solver example, and Java provides the necessary methods for communication.

The code computes a fixed number of square roots, and it splits the work among a variable number of threads. These threads are then mapped to the processors by the operating system, relieving the user of the need of employing any kind of message-passing library as well as a load-balancing algorithm. The code runs on any kind of platform as long as Java is available.

The purpose of this code is to determine whether multithreading gives parallel speedup on the target parallel machine.

3.2. Matrix multiply

Parallel matrix multiplication is implemented by block matrices, as shown in Fig. 1. Matrices **A** and **B** are multiplied to produce **C**. In the figure, we see how computation of each sub-block of **C** requires concurrent read access to matrices **A** and **B**, but not for writing into **C**. In the benchmark suite are actually two programs: one being the multithreaded version, the other being, for comparison, a serial version that does not use threads. In addition, we have a C-coded version of the sequential block-matrix multiply. The purpose of this component of the test suite is:

- to compare floating-point performance for scientific applications between C and Java on the HP V-class, the

ClibPDF - www.fastio.com

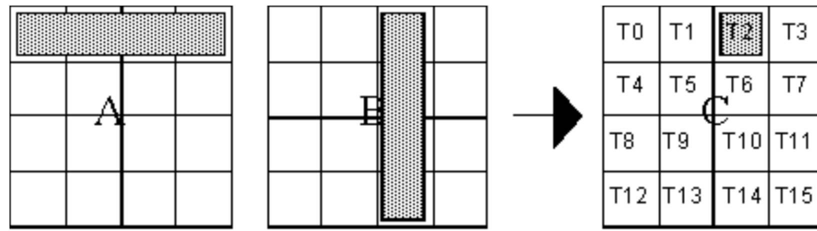

Fig. 1. The multithreaded matrix multiplication is performed by splitting matrix C into partitions. Each partition is then calculated by one thread, with the thread numbering as shown for matrix C. Concurrent access to the memory containing A and B is necessary; here we see the memory that thread 2 accesses.

Sun Enterprise 450 (Ultra SPARC II, 300 MHz) and also a Pentium II (300 MHz, Linux);
- to measure parallel efficiency of a multithreaded application that has some read contention.

3.3. Mandelbrot

This code tests the self-scheduling of threads. Computation of the famous Mandelbrot set utilizes a 2D grid in the complex plane, and an independent iterative calculation takes place at each grid point where the number of iterations varies greatly from point to point. We partition the grid into blocks; we want each block large enough that thread overhead will be much less than the computational work associated with the block, and we want the blocks small enough that there are many blocks for each processor. As explained above, each processor takes a block from the pool, computes it, then gets another block.

Although this program is still embarrassingly parallel, it exhibits a new feature, namely, that computational load depends on the position within the solution domain, which is a rectangle in this case. Dynamic load balancing would be needed to run such an application successfully on a large parallel architecture. Using PVM or MPI, the user has to provide a sophisticated algorithm to achieve this feature requiring a lengthy piece of code. Using the Java thread concept, dynamic load balancing is provided by the operating system.

The parallel Mandelbrot concept is simple. There is a nonlinear mapping in a finite region of the complex plane (rectangular). Since the mapping can work on any finite region without interference with any other region being computed, the parallelization strategy is a simple, 1D-domain decomposition. This is as far as the idea goes. The final threaded code, however, shows a fair degree of complexity. This example can be used to demonstrate the effect of dynamic load balancing achieved by the Java thread concept.

3.4. A parallel Laplace solver using Java threads

The 2D solution domain is subdivided into a specified number of 2D subdomains (variable according to user input) that are computed independently of the other subdomains. The concept of ghost or halo points, as shown in Fig. 2 is used to model the overlap between neighboring blocks. These subdomains have to communicate after each iteration step to update their boundary (ghost) points. In this regard, parallelization is simply done by introducing a new (interblock) boundary condition. However, this condition was already present in the sequential code because complex geometries had to be modeled by using the multiblock concept. It should be noted that the multiblock concept,

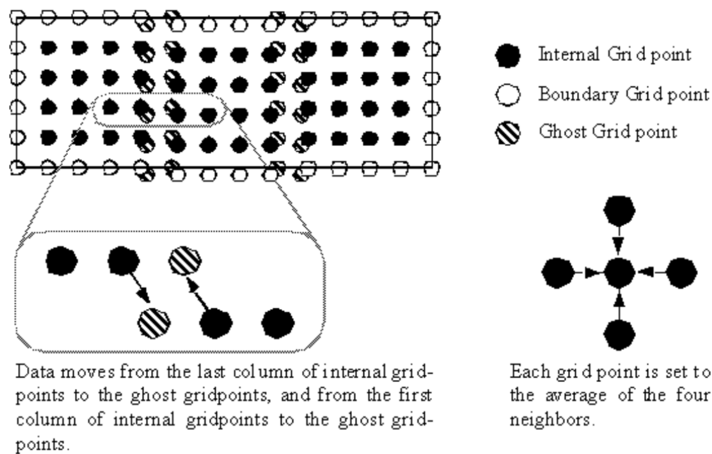

Fig. 2. Overlap or ghost points for a Laplace solver. An overlap of one column or one row is used. It should be noted that this overlap size is not sufficient for higher-order numerical schemes. An overlap of size two is used for the flow solver *ParNSS* and *JParNSS*. The Java Laplace solver is described in Section 3.4.

ClibPDF - www.fastio.com

besides numerical advantages, is not subject to the severe performance reduction caused by frequent cache misses.

It is also interesting to compare the message passing Laplace solver written in C with the thread-based Java Laplace solver. Parallelization using explicit messages is much more elaborate and far less readable, while in Java we simply extend a class from Thread and form new instances of this class in a simple *for* loop using the *new* operator. Since the class is derived from *Thread* it automatically inherits all the methods of this class, including the communication methods.

Subdomains or blocks are connected via edges, but not via vertices. That means, communication takes place only across edges, but not across diagonals. Hence, each block is connected to at most four neighbors in 2D (six neighbors in 3D). If only first derivatives have to be computed numerically, diagonal points are not needed. However, for second derivatives the computational stencil needs these diagonal values. Since communication does not take place across diagonals these values are not explicitly updated. Therefore, a different computational stencil has to be used that computes the missing value from its neighbors, omitting the diagonal value. The scheme retains the same numerical order but the truncation error changes. For instance, if viscous terms have to be computed in the Navier–Stokes equations, this practice has shown to be both accurate and effective. In this regard, there is a minor difference between the sequential and the parallel numerical algorithms.

The major difference to an irregular problem is that all blocks are oriented with respect to a single coordinate system and a regular structure is present. The regular problem approach, of course, can be extended to curvilinear coordinate systems, but in the computational space there would always be a regular structure, i.e. a multidimensional cube or parallelepiped.

3.5. Ghost points

To parallelize the Laplace solver, see also [13,14], we split the rectangular set of gridpoints into rectangular subgrids, and each node is responsible for one of these subgrids, as shown in Fig. 2.

However, at the edge of one of these domains, it is necessary to read the value of the field from a grid point which is in another node. To do this, we have established a cache of these values on each side of the boundary; in addition to the internal points and the boundary points, this cache is known as *ghost (or overlap) points*. A ghost point in one node may be (conceptually) thought of as lying in the same place as an internal point that is stored in another node; the ghost point value should follow the value in the internal point.

In the sequential code, we do the Jacobi relaxation repeatedly until convergence; with the parallel code the loop alternating between updating the ghost points and doing the Jacobi relaxation.

3.6. Java parallel framework: client–server

In this section we will describe a general, complete Java parallel framework for solving problems in science and engineering. The approach is based on the client–server concept and thus allows to perform a parallel computation using the Internet. The code comprises three parts: namely the client, the server and the shared part. The shared part is a Java *interface* that has to be implemented by either the client or the server. First, all codes have to be compiled to generate the class (Java bytecode) files. This can be done using any Java compiler on any machine. On the server, the *rmic* [15–17]—the rmic is part of the Java Development Kit (JDK)—compiler has to be evoked to produce the *stub* and *skeleton* classes. The stub class along with the client code resides on the client computer, and the skeleton class is on the server machine. The communication between client and server takes place through these two objects. Next, the so called *rmiregistry* is started on the server to register all objects that can perform communication. The *rmiregistry* command is used for this purpose. In general, the registry now is ready to communicate over port 1080 and listens to communication requests. In the next stage, the server code is started and the objects for communication are actually registered. When the server is started an address is supplied in form of an rmi address, i.e. *rmi://hostname/RMIObject* where hostname is the name of the server. The RMIObject name can be any name, but it must be the same for both client and server. We used JParFW. A domain name service (DNS) must be enabled that translates this name into a valid IP address. In the last step, the code on the client is started using the same rmi address. Since both client and server know the shared interface code that contains an interface for the *JPSolver* that is implemented on the client side, the client can send over its own solver object at run time. The server knowing the interface of the JPSolver object, therefore, has the necessary information about the signature of all solver methods (in non object-oriented terminology methods are referred to as functions) and thus knows how to handle the solver object. The Java parallel framework does not know anything about the numerics or physics that is actually implemented. It provides, however, the necessary parallel infrastructure for all solver objects that implement the JPSolver interface. Hence, parallelization is done once and very different solver objects can be constructed, resulting in a parallel code that might solve problems in different fields of science and engineering. The solver used as an example is the Laplace solver described in the previous section.

The parallel framework code, JParFW, has been extracted from the parallel Java flow solver JParNSS [6] that is currently being developed. The latter code uses many features like user authentication, multiple session manager and class loaders that are not present in the JParFW. JparFW, however, comprises only 800 lines of Java instead of the 20,000 lines of the JParNSS code.

ClibPDF - www.fastio.com

Table 1
Computing times and speedups for square root program on HP V-Class. Essentially linear speedup is achieved for up to 32 processors. The same behavior holds for the 4-processor Sun

Number of threads	Time (s)	Speedup
1	12:41	1.00
2	6:34	1.93
3	4:20	2.92
4	3:17	3.86
8	1:39	7.69
9	1:28	8.65
16	0:50	15.22
32	0:26	29.27

3.7. JParEuler

This code solves the Euler equations with a first-order explicit method, using a simple multiblock grid. The grid is a collection of rectangular blocks, each with a regular, square mesh. This code has client–server control through Java RMI.

It also has a larger ratio of computation to communication than the Laplace example.

4. Results

The main goal of the present Java test suite is to investigate parallel scalability for the thread-based parallelization approach. The numerical performance is also important, but will definitely improve with new compiler releases, and therefore is not our major concern.

4.1. Square root

In this simple program, we expect essentially linear speedup as the number of processors is increased. Or, if this example cannot use multiple processors efficiently, there is no hope of achieving a parallel Navier–Stokes solver, since such a code requires a substantial amount of communication at each iteration.

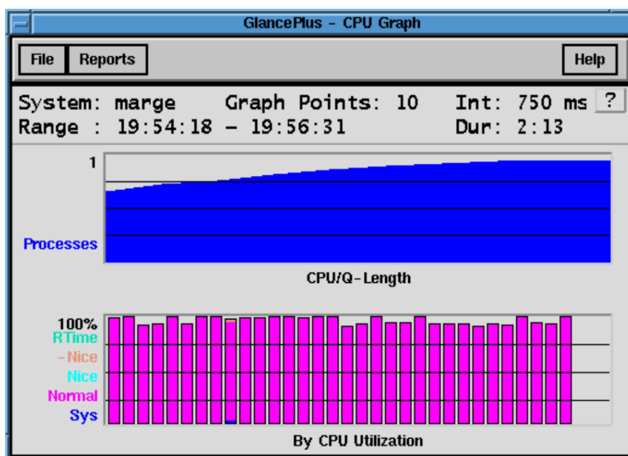

Fig. 3. The program GlancePlus is used on the HP V-Class to illustrate the parallel runtime behavior for the square root program. The upper graph shows a history of the averaged machine load. The lower graph shows the CPU utilization. The workload of all CPUs is close to the ideal 100% level, resulting in excellent speedup.

Table 1 shows that the HP V-class provides essentially a linear speedup for the Java threaded square root example, with an efficiency of over 90% on 32 processors.

Fig. 3 shows the CPU usage depicting saturation of all processors within the first few seconds of the run.

Table 2
Comparison of the single-processor floating-point performance for a sequential matrix multiplication algorithm between C and Java on HP V-Class, Sun Enterprise 450 (Ultra SPARC II, 300 MHz) and Pentium II (300 MHz, Linux)

Hardware and software specification	MFlops per second for different matrix sizes		
	30 × 30	100 × 100	300 × 300
HP V-Class, C code[a]	242.00	237.00	114.00
HP V-Class, Java 1.1.7	9.33	9.57	9.54
SUN E450, C code[b]	176.86	157.73	35.24
SUN E450, Java 1.1.7	6.35	6.72	5.87
SUN E450, Java 1.2	17.08	12.65	8.90
Pentium II, 300 MHz, Linux C code pgcc/egcs Pentium compiler group	90.00	91.74	39.82
Pentium II, 300 MHz, Linux, Java IBM 1.1.6, Jitc	24.80	22.79	11.21

[a] Optimization level: +O3.
[b] Optimization level: fast.

4.2. Matrix multiplication

4.2.1. Sequential matrix multiplication

Table 2 shows the performance, in megaflops, of the sequential matrix-multiply program on one processor of the HP-Vclass, the Sun E450 and a Pentium II PC running Linux.

The performance of the C code decreases significantly with the size of the matrix. We interpret this as a caching effect—in a modern computing system the CPU is rarely a bottleneck, even for numerically intensive code. Rather, it is the memory subsystem that usually cannot deliver data fast enough to the CPU. For the smallest matrices the performance ratios are:

- for the HP, C is 26 times faster than Java 1.1.7;
- for the Sun, C is 28 times faster than Java 1.1.7, and 10 times faster than Java 1.2;
- for the Pentium, C is 3.6 times faster than Java 1.1.6.

The performance of Java 1.2 is nearly three times better than the old Java 1.1 on the Sun architecture. We expect the same kind of improvements when HP releases its Java 1.2 implementation (late 1999). We would expect further large

ClibPDF - www.fastio.com

Table 3
MFlop rates for multithreaded matrix multiplication algorithm. Columns are labeled with the size of the matrices

Number of threads	HP V-Class using 16 processors[a]			HP V-Class using one processor[b]			Sun E450[c] using four processors		
	30 × 30	100 × 100	300 × 300	30 × 30	100 × 100	300 × 300	30 × 30	100 × 100	300 × 300
1	7.01	8.67	8.64	6.51	8.77	8.66	13.40	13.47	9.06
4	11.38	31.35	33.49	3.86	8.34	8.68	19.21	28.60	23.56
9	6.33	–	72.53	2.40	–	8.73	12.25	–	22.00
16	–	65.40	118.68	–	7.74	8.69		27.01	28.13
25	2.62	59.17	112.97	1.04	7.28	8.65	5.14	42.21	27.84
36	1.83	–	110.66	0.75	–	8.64	3.75	–	30.07
100	0.64	22.84	109.53	0.29	4.82	8.44	1.57	29.43	33.93

[a] Java version "HP-UX Java C.01.17.00 99/02/08"; mpsched-l 1 java Test a b c.
[b] Java version "HP-UX Java C.01.17.00 99/02/08"; mpsched-c 20 java Test a b c.
[c] Solaris VM (build Solaris_JDK_1.2_01_dev06_fcsV, native threads, sunwjit).

speed increases when a new Java compiler is released by HP (early 2000). The Java speed for the Pentium processor seems to be in a much more advanced state, and with the further development of Sun's hotspot technology, we expect the speed of Java programs to rival the performance of C codes.

4.2.2. Multithreaded matrix multiplication

Table 3 shows megaflop rates for the pure Java multithreaded matrix-multiply benchmark. The table shows absolute performance for the HP and the Sun machines, and also gives a comparison between one and 16 processors of the HP architecture. A maximal speedup of 13.74 for 16 processors for the 300 × 300 matrix example was measured.

For the 16-processor HP V-class results, we notice a performance increase for the 300 × 300 matrix example, with the number of threads, up to a limit of 16. After that, the performance slightly decreases because of additional thread administration time, since for the given problem size and 16 threads the machine is saturated. The same is true of the 4-processor Sun. But this is not true for a smaller matrix size: performance increases only a little bit and then decreases. Here, the overhead in creating threads dominates the computational work of the processor.

Missing values in the table are caused by implementation strategy, requiring all block matrices to be of the same size, so that the number of threads must divide the number of matrix elements.

4.3. Mandelbrot

This part of the benchmark suite tests the management of the thread pool. In previous sections, we have discussed multithreaded codes where threads are created at the beginning of the code, then run until the end of the code. In this example, however, processors take, run and stop threads repeatedly during execution. A bottleneck in the dynamic allocation of threads to processors would thus be found in this part of the test suite.

Fig. 4 shows CPU utilization when the Mandelbrot examples is run with seven processors, and it can be seen

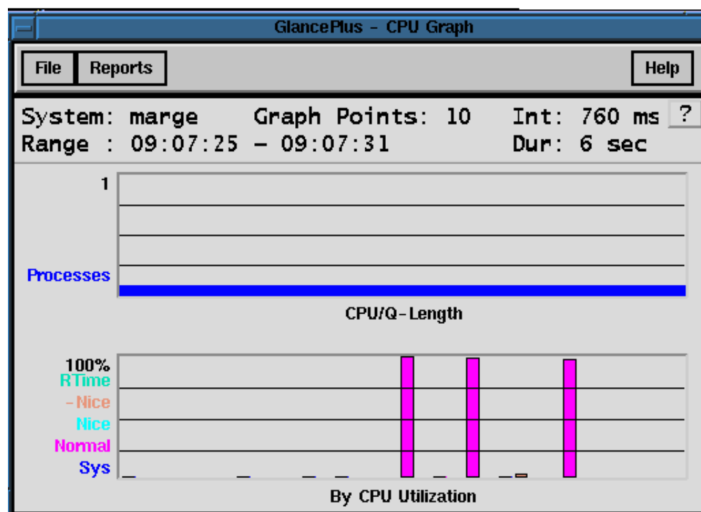

Fig. 4. CPU Graph for Mandelbrot computation using seven threads on HP V-Class. Only three processors are used, although no communication of synchronization is required.

ClibPDF - www.fastio.com

Table 4
Computing times and speedups for Laplace program on 4-processor Sun using 192 threads (16 × 12 blocks with 9600 cells) running 5000 iterations. We achieve in this test a 50% CPU load only

Number of processors	Time (s)	Speedup
1	893	1.00
2	639	1.40
3	562	1.59
4	578	1.54

that only three processors are actually in use. The timings seem to be very inconsistent, and we are working with HP to investigate the problem.

4.4. Laplace

In this test program we are using global thread synchronization instead of lose thread synchronization. This means that all ready threads are waiting until the last thread becomes ready, then the next iteration can be computed. With this synchronization method we achieved only about 50% CPU load (Table 4). With a lose synchronization, as we are using in the JParEuler test program, we have a much better use of the available resources ~100% CPU load. In the lose synchronization model a thread is only waiting until its neighbor threads become ready, so the balance between running and waiting threads during the computation is improved.

4.5. JParEuler

As a test case, we have chosen to compute an Euler flow past a forward-facing step at Mach 3. The resulting Mach number field is shown in Fig. 5a. A strong bow shock is formed, with an expansion fan radiating from the corner, and a cascade of shock reflections downstream.

The computational grid has (at minimum) three rectangular blocks that surround the step-corner. Fig. 5b shows splittings of these elemental blocks into a total of 16 and 48 blocks. It is these that are used for the actual computation, because it is possible to use many more processors when there are many more blocks. Six different grids have been created with topologies shown in Fig. 5b, three with 16 blocks and three with 48 blocks. For each of these grids, all of the blocks have the same number of gridpoints, and therefore we expect them to have the same computational work. Furthermore, the ratio of number of blocks to processors is an integer, and so the theoretical parallel efficiency is 100%.

The results are shown in Table 5 where 320 iterations of the first-order explicit Euler scheme have been timed.

When there is more than one thread per processor, i.e. the 48-block runs, then we get much better speedup, which is because there is enough computational work to keep each processor busy. We also notice that when the number of cells per block increases, then the corresponding speedup decreases.

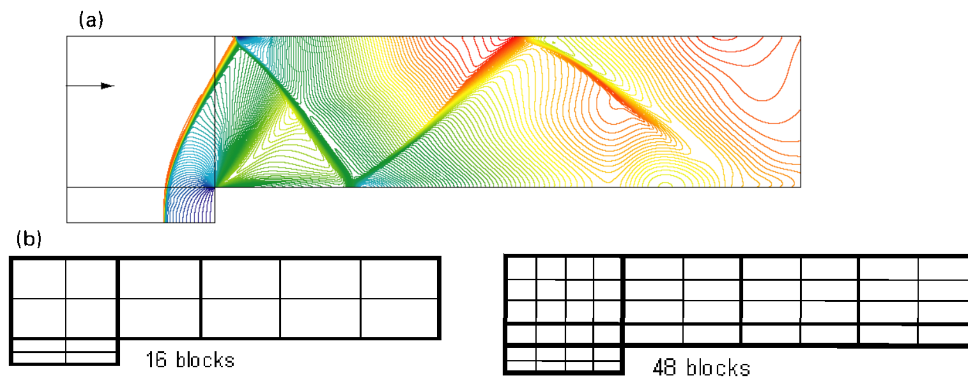

Fig. 5. (a) Euler flow past a forward-facing step at Mach 3. The computations are explicit and first-order accurate. Shown is the Mach-number distribution. (b) Splitting of the grid for the forward-facing step into 16 and 48 blocks.

Table 5
JParEuler on HP V-Class. Times are for 320 iterations

Number of blocks	Number of cells	Time (s)		
		Single processor	Multi processor (16)	Speedup
16	121,104	3246.73	541.13	6.00
16	200,704	6908.88	1077.20	6.41
16	484,416	12905.88	2720.48	4.74
48	118,803	2980.93/	225.76	13.20
48	202,800	5190.54	436.09	11.90
48	480,000	12663.30	1162.54	10.89

ClibPDF - www.fastio.com

Table 6
JParEuler on SUN E450, with 320 iterations

Number of blocks	Number of cells	Time (s)		
		Single processor	Multi processor (4)	Speedup
16 (B10)	121,104	852.79	258.26	3.30
16 (B11)	200,704	1571.775	532.74	2.95
16 (B13)	484,416	4277.962	1593.45	2.68
48 (B6)	118,803	756.24	195.95	3.86
48 (B9)	202,800	1409.962	378.61	3.72
48 (B8)	480,000	3892.67	1077.06	3.61

Table 7
JParEuler on SUN E450, with 1000 iterations

Number of blocks	Number of cells	Operating system	Time in s multi processor (4)
48 (B3)	42,000	Solaris 2.6	282.04
48 (B3)	42,000	Solaris 7	258.16

Table 6 shows the results from the Sun machine, where the speedup is between three and four for all of the configurations.

4.6. Comparing Solaris 2.6 and Solaris 7 using exactly the same JDK release (build Solaris_JDK_1.2.1_03, native threads, sunwjit) on a Sun Enterprise 450

For this comparison we also use the JParEuler test to demonstrate the influence of the operating system on the computation while using the same JDK release. Table 7 shows that Solaris 2.6 takes about 24 s more for the same computation than Solaris 7. A reason for this behavior is that the internal Thread (LWP) handling of Solaris 2.6 takes much more time doing an efficient load balancing than Solaris 7 (Fig. 6) since this OS, Solaris 7, is a 64 bit system. To have no side effects on using 64 bits and to have a better kernel architecture comparability vs. Solaris 2.6 (32 bit only) we started the Solaris 7 explicit in a 32 bit mode.

5. Conclusions

The main emphasis of the Java test suite has been to investigate the parallel efficiency of the Java thread concept. Up to 32 processors have been used. As a result, we conclude that parallel efficiency is obtained if a sufficient number of threads and sufficient computational work within a thread can be provided. With the scientific and engineering problems that we have in mind, in particular fluid dynamics for complex geometries, these requirements are easily satisfied. For instance, calculating the flow past a 500 block X-33 configuration or a 4000 block Ariane 5 launcher utilizing several million grid points, clearly allows us to use hundreds or even thousands of processors. In this regard, we feel safe to say that parallel efficiency can be achieved via the thread concept and thus large-scale parallel applications in Java are possible.

On the other hand, the speed of the Java code generated by the compilers of the main hardware vendors is unsatisfactory. However, these compilers are in an early stage and something substantial can be expected within the next 18 months. The IBM Alpha Works compiler delivered good results and we see no principal reason that Java code should be significantly slower than Fortran or C code. In

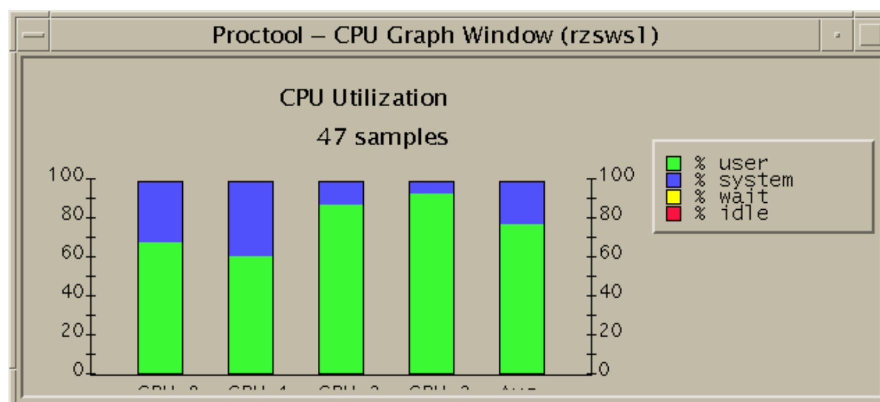

Fig. 6. CPU Graph during the first minute of the JParEuler computation on Sun E450 using Solaris 2.6. The system requires ~20% of the CPU resources for the internal Thread (LWP) handling.

ClibPDF - www.fastio.com

addition, the new Solaris release has shown that the OS itself can decrease computing time by more efficient thread handling.

We are, therefore, not concerned by the speed issue; and leave this problem to the compiler builders. The major point for large-scale applications is parallel efficiency and the results using the thread concept have been very encouraging. Further work will be needed, but we follow Kernighan's rules, *Make it right before you make it faster* and *Don't patch bad code, rewrite it*, the latter rule being the reason for a pure Java flow solver code. The Java OOP approach and the unique Internet capabilities through RMI provide a major advantage over all other existing programming languages.

Acknowledgements

This project was partly funded by the Ministry of Science and Culture of the State of Lower Saxony, Germany and the European Commission under contract JavaPar 1998.262. We would like to thank the Center for Advanced Computing Research at Caltech for hosting R.W. while this work was completed. This work contains part of the PhD work of T.L. J.H. is grateful to Professor Rolf Jeltsch, ETH Zurich for being his host during his sabbatical leave at the Department of Mathematics, 1999, where part of this work was done.

In the writing of the Java test suite the books by C. Horstman [1,4] were most useful, since they demonstrate in a masterly way the full capabilities of this language.

References

[1] Horstman CS, Cornell G. Advanced features, CoreJAVA 2, vol. II. Englewood Cliffs, NJ: Prentice Hall, 2000.

[2] Lea D. Concurrent programming in Java. Reading, MA: Addison Wesley, 1997.

[3] Gosling J, McGilton H. The Java language environment—a white paper. Sun Microsystems, http://www.javasoft.com/docs/, 1995.

[4] Horstman CS, Cornell G. Fundamentals, CoreJAVA 2, vol. I. Englewood CLiffs, NJ: Prentice Hall, 1999.

[5] Fox GC, editor. Java for computational science and engineering-simulation and modeling. I. Concurrency practice and experience, vol. 9(11). New York: Wiley, 1997.

[6] Fox GC, editor. Java for computational science and engineering-simulation and modeling. II. Concurrency practice and experience, vol. 9(11). New York: Wiley, 1997.

[7] Häuser J, Williams RD. Strategies for parallelizing a Navier–Stokes code on the intel touchstone machines. International Journal for Numerical Methods in Fluids 1992;15:51–58.

[8] Winkelmann R, Häuser J, Williams RD. Strategies for parallel and numerical scalability of CFD cods. Comparative Methods in Applied Mechanical Engineering. vol. 174. Amsterdam: Elsevier, 1999. p. 433–56.

[9] Moreira JE, Midkiff SP, Gupta M. From Flop to Megaflop: Java for technical computing, IBM Research Report RC 21166.

[10] Moreira JE, Midkiff SP, Gupta M. A comparison of Java, C/C++, and Fortran for numerical computing, IBM Research Report RC 21255.

[11] Häuser J, Williams RD, Spel M, Muylaert J. ParNSS: an efficient parallel Navier–Stokes solver for complex geometries. AIAA 94-2263, AIAA 25th Fluid Dynamics Conference, Colorado Springs, June 1994.

[12] Häuser J, Ludewig T, Gollnick T, Winkelmann R, Williams RD, Muylaert J, Spel M. A pure Java parallel flow solver. 37th AIAA Aerospace Sciences Meeting and Exhbition. AIAA 99-0549, 1999. p. 11–14.

[13] Proceedings of the ACM Workshop on Java for High Performance Network Computing, Stanford University, Palo Alto, CA, U.S.A. 28 February–1 March, www.cs.ucsb.edu/conferences/java98, 1998.

[14] Java Lapace implementation, from AppStar LLC., http://www.appstar.com.

[15] Sun Microsystems, Java remote method invocation specification, revision 1.41, JDK 1.1.1, 24 March, http://www.javasoft.com/docs/, 1997.

[16] SunSoft, Java on Solaris 2.6—a white paper, http://www/sun.com/solaris/Java/wp-Java/, 1997.

[17] Java Party, an improved remote method invocation, University of Karlsrühe, www.appstar.com.

ClibPDF - www.fastio.com