Large-Scale Parallel Geophysical Algorithms in Java: A Feasibility Study

Matthias Jacob* and Michael Philippsen
Computer Science Department
University of Karlsruhe, Germany
[jacob|phlipp]@ira.uka.de

Martin Karrenbach
Geophysical Institute
University of Karlsruhe, Germany
martin.karrenbach@physik.uni-karlsruhe.de

Abstract

Java is often accused of being too slow for serious programming, especially for scientific problem solving. However, we found that for a large-scale geophysical application, Java code compiled with current just-in-time compilers runs slower than Fortran by a factor of at most four, on both a shared-memory parallel machine (SGI Origin2000) and a distributed-memory parallel machine (IBM SP/2). The moderate slow-down is easily offset by the following advantages: (a) object-oriented Java code is easier to maintain and reuse than Fortran code, (b) Java code is fully portable, even among parallel computers with different memory models. Furthermore, better compiler technology is on the horizon, which will narrow the performance gap even more.

1 Introduction

Within Geophysics, seismic methods are an essential tool for petroleum and gas exploration. They produce images of the earth's interior and let explorationists analyze the geological structure of the underground. Seismic methods process data obtained from sound wave reflections caused by different structural elements within the earth. Sound data is analyzed with respect to the variation in time delay of the reflected signal. The necessary operations demand vast amounts of processor time and memory.

The implementations of such methods in procedural programming languages like FORTRAN that are largely used in petroleum industry have disadvantages:

- Procedural programming languages usually lack in software reusability and therefore make it difficult to maintain the source code. They don't involve an intrinsic hierarchy of operators, needed for a compound framework of seismic methods.

  With the main focus on object-oriented languages the GOON/CLOP architecture [1, 10] for implementing seismic methods in C++ has been developed within the Stanford Exploration Project. Although GOON/CLOP is appropriate for arranging operators for reusability, it suffers from performance problems, and it is not capable of being parallelized easily.

- Message Passing libraries are used to implement seismic methods on parallel distributed memory computers. With these libraries, the programmer does not only have to deal with the seismic problem but in addition has to tackle the problems of parallel programming: load balancing and locality. The resulting program code is not only error-prone but cannot easily be ported from one machine architecture to another.

In this situation, the Java programming language [3, 4] has been introduced. Java is object-oriented and offers language elements to express thread-based parallelism and synchronization without additional system libraries. In combination with the integrated Remote Method Invocation (RMI) of Java, data exchange between processors is easy. This allows parallel processing on application level, i.e., in the Java environment itself, as Hassanzadeh et al. discuss for geophysics [5]. There is no further need for Message Passing or virtual shared memory within the application. Additionally, this parallelism is hardware independent, i.e., programs run on any major platform.

From the software engineering point of view, Java seems to be a perfect programming language for the parallel implementation of seismic methods. Unfortunately, Java has the reputation of being slow and RMI of being even slower. In this paper we show that Java can achieve acceptable performance for scientific

*Part of this work has been supported by a diploma thesis grant of the Prof. Dr.-Ing. Erich Müller-Stiftung, Essen, Germany.

computing.

For this study we picked an operator according to the specific area of velocity analysis, namely the Veltran operator. The Veltran operator is a basic method in seismic processing. Efforts are underway at the Stanford Exploration Project to produce a seismic operator framework (JAG [12]) in Java. We base our implementation of a parallel geophysical operator on a preliminary version of JAG. After a short introduction into this geophysical operator in section 2, the architecture of the underlying distributed runtime environment, JavaParty, is discussed in section 3. In section 4 the parallel implementation of the operator is sketched. The final section gives the results of our measurements.

2 The Veltran operator

The Veltran operator for velocity analysis is one of the elementary seismic methods. It is the first of a pipeline of operators used for the analysis of the earth's subsurface layers and is therefore directly applied to the data obtained by seismographic devices. Such measurements are subject to reflection properties of the earth's layers.

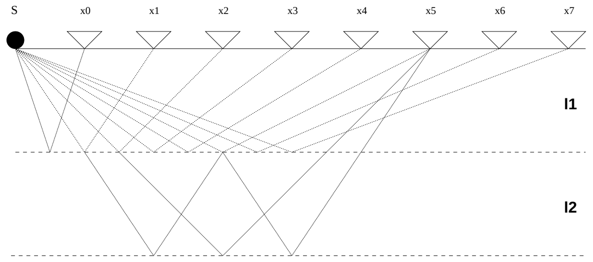

Figure 1: Measurement of seismic data – the waves emanate at s and x_0-x_7 are distinct seismographic devices. The dotted line is the boundary between two horizontal layers l_1 and l_2 with different specific velocities.

As shown in Figure 1, a wave is generated at a distinct point s and expands circularly. If the media within two horizontal layers namely l_1 and l_2 differ in their specific velocity, a part of the incoming waves are reflected. The energy conveyed by these reflected waves can be measured with seismographic devices at distinct points $x_0 - x_7$. The Veltran operator takes the signal readings of the seismographic devices and derives the specific velocities. For these purposes a simplified model of the earth is assumed where the layers are arranged horizontally.

From the input data, we have the measured signal travel time t and the distances between s and x_i. Each of the two graphs in Figure 2 contains a two-dimensional matrix of incoming data, called an input plane, that describes the power of the signal received by the seismographic devices over time. Mathematically, the travel time curves can be arbitrarily shaped, but for horizontal layers the curves are hyperbolic. The first graph uses synthetical data to show the hyperbolae more prominently, whereas the second graph depicts real data.

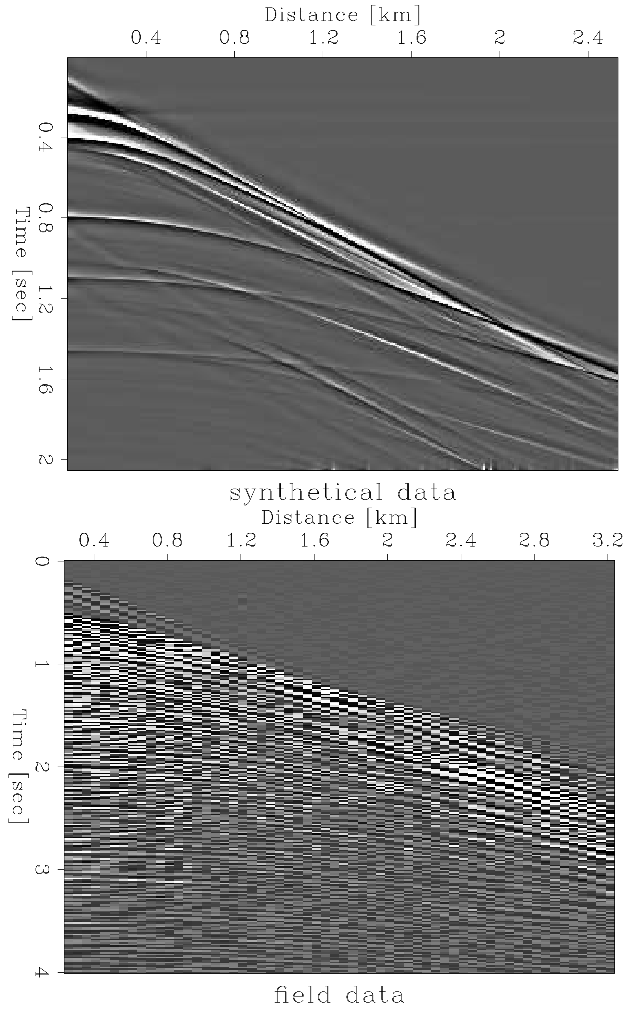

Figure 2: Source data illustrating the hyperbolae: (a) synthetical data, (b) field data (courtesy Mobil Oil). Time relates to τ (see below) and distance represents the distance between s and x_i.

If in addition, we knew the time τ consumed for the signal propagating straight down from s to the layer and back, the following formula, called "normal moveout", could be used for computing the velocities:

$$t = \sqrt{\tau^2 + \frac{x^2}{4v^2}}$$

Unfortunately, τ cannot be measured since it is impossible to install a seismographic device directly at s. Therefore, Veltran must infer the layer velocity and pseudo-depth from the data indirectly.

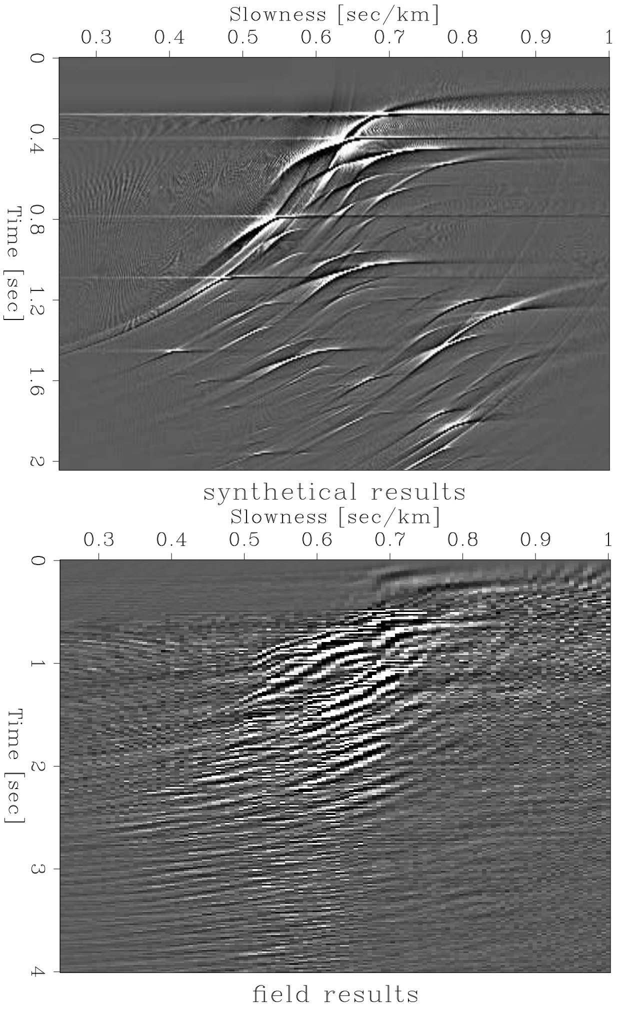

Figure 3: Veltran results for input from Figure 2 illustrating the maxima of the amplitudes: (a) synthetical data, (b) field data (courtesy Mobil Oil). Time relates to τ and slowness describes the reciprocal of the velocity v.

This is done as follows. Veltran iteratively assumes potential velocities from a range of n_v values. For each of these velocities, hypothetical hyperbolae are computed. The signal measurements from the input matrix are added along the hyperbolae. For the correct velocity the hypothetical hyperbolae and the actual hyperbolae in the data match – the sum of the measured signals along the hyperbolae reaches a maximum. The computation of the sum takes n_t steps since each time step must be considered. Hence, for a single input plane, Veltran takes $n_v \cdot n_x \cdot n_t$ steps.

Figure 3 shows the result of Veltran: The brighter a point is for an assumed velocity, the higher is the sum of the measured signal readings along the hyperbole that corresponds to that velocity. The brightest point in each row refers to the specific velocity of the layer with the time offset τ. Some unavoidable side-effects often make more than one point look bright. This is because a part of the waves can be reflected more than once, causing distortion in the projected results.[1] To reduce the strength of such effects, additional techniques are used that are beyond the scope of this paper. These typically involve iterative linear and non-linear solvers such as the conjugate gradient method. For more details see [2, 13].

For reconstructing the a 2D image of the earth's interior, measurements for more than just one source location are necessary. The most common way to obtain the data is to move the point of wave generation along a line on the earth's surface, producing several input planes, i.e., a 3-dimensional input matrix. In addition to $n_v \cdot n_x \cdot n_t$ steps needed for Veltran on a single input plane, Veltran now has to take the third dimension with n_s iterations into account.

For obtaining a 3D model of the earth's interior seismographic devices and emanation points are positioned within a surface area instead of a single line. Hence input data becomes 5-dimensional. But these additional dimensions are independent from the computation model since the measurements within the surface area can be transformed to a single line. This is done by concatenating the lines of the area to a straight one. Therefore, only the lengths of the s- and v-axis increase.

3 JavaParty - the distributed runtime environment for Java

JavaParty [8, 11] is a programming layer on top of Java and RMI. JavaParty inherits the advantages of both, but it is designed to avoid their disadvantages.

Java's threads and synchronization mechanisms offer appropriate means for parallel programming of shared memory parallel computers. Unfortunately, Java does not provide elegant and straightforward mechanisms for parallel programming of distributed memory machines, e.g., of an IBM SP/2 or a cluster of workstations.

On these platforms, the programmer can either use explicit socket communication to connect between various nodes or use Remote Method Invocation (RMI). Both approaches, however, result in complicated and

[1] In Figure 1, one of the waves has been reflected twice.

error-prone programs, which have similar disadvantages as programming with message passing in general:

- Depending on communication requirements, multi-threaded Java programs often grow by about 70% when being ported from a shared memory environment to a distributed memory platform by means of sockets or RMI [11]. Both the socket and RMI mechanisms are complicated. They induce a significant extra amount of implementation work and render the programs more difficult to understand and maintain.

- Although the source code of message passing programs with hard-wired data and thread distribution is often portable between architectures, porting frequently causes significant performance losses [6, 9]. Moreover, it is often difficult to adapt programs with explicit communication for varying numbers of nodes, for different architectures, or for changing network topologies.

JavaParty transparently adds remote objects to Java purely by declaration and avoids exposing the programmer to sockets, RMI, and message passing libraries. JavaParty code is preprocessed into Java code with RMI hooks, both are then compiled by regular Java and RMI compilers into platform independent and secure ByteCode.

JavaParty extends Java with a new class modifier `remote`. By this modifier, the programmer can distinguish between objects that are local and objects that may be instantiated on a remote node. Since Java's threads are implemented by means of objects as well, the programmer can create remote threads that run on remote processors. JavaParty implements Java's object semantics, i.e., the programmer has the impression of writing regular multi-threaded Java programs. The source code size does not change when moving from Java to JavaParty, but JavaParty programs are portable between single processor workstations, shared memory parallel computers, and distributed memory platforms. Since the topology and the number of processor nodes of the underlying parallel computer is completely transparent in JavaParty programs, programs written in JavaParty automatically adapt to changing configurations.

When it comes to performance, locality is the most crucial issue of programs for distributed memory computers. Because of cache hierarchies, locality is significant for the shared memory architectures as well. Lack of locality induces communication through the interconnection network of the distributed memory platform or results in cache misses on shared memory architectures. Both are at least an order of magnitude slower than local accesses and must be avoided wherever possible.

JavaParty provides compile-time and run-time support for achieving locality. When remote objects are instantiated or remote threads are started, JavaParty's object distributor automatically assigns remote threads to processor nodes to achieve good load balancing. Moreover, JavaParty's object distributor tries to assign all objects that are used by a remote thread to the same node. If different threads use the same data objects, the object distributor implements heuristics that keep the amount of node-to-node communication small. Since different phases of a program may need certain objects on varying nodes to achieve high-frequency local accesses, JavaParty monitors object access patterns and migrates objects to different nodes at runtime.

The programmer is free to overrule the decision of JavaParty's object distributor and to place remote objects and remote threads manually. Analogously, the programmer can disable automatic object migration and replace it with explicit migration commands. Moreover, the programmer can even implement application-specific object distributors and migration managers. The implementation of the Veltran operator, shown below, uses JavaParty's standard object distributor.

The JavaParty preprocessor and the rest of the JavaParty environment are 100% pure Java and freely available [8].

4 Parallelizing the Veltran Operator

An efficient parallelization of Veltran is needed since Veltran runs long and consumes a vast amount of memory.

Since the velocities of different planes can be computed independently and result in a trivial parallelization we consider a single input plane for a single emanation point first.

The main problem for parallelizing Veltran is the non-linear structure of the hyperbolae. For a hypothetical velocity v that is to be considered during the Veltran iterations, all values from the $t \cdot x$ array might contribute to the computation of the maximum, i.e., the velocity.

Therefore, when the computation is spread to the processors along the x axis, every processor computes a matrix of partial sums. In a completion phase, all parts of the results have to be summed before getting the final output of this operator. The completion phase would require an additional amount of $n_t \cdot n_v$ operations[2] which extends the total runtime of the operator considerably.

The parallelization along the v- or t-axis avoids the complexity of the completion phase and is hence more appropriate. If every processor analyzes some of the hypothetical values of v, every processor computes a segment of the resulting matrix without interference from other processors. The resulting matrix is thus produced by merging the distinct result matrices together. The same argument holds for the t-axis.

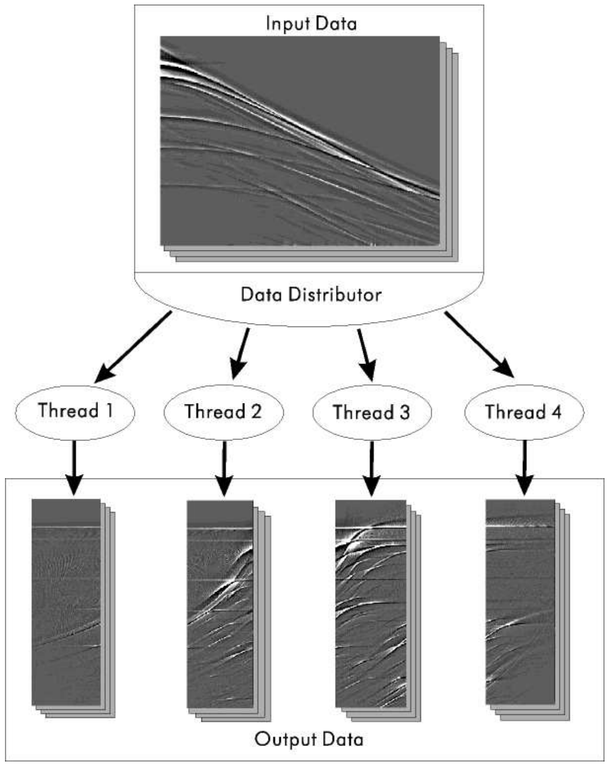

Figure 4: Architecture of the parallel implementation of the Veltran operator. The Data Distributor divides input data along the v-axis. Concurrent threads compute separate parts of the resulting arrays that can be merged together.

The parallel version of the Veltran operator is a combination of remote and local components. The initial object is started on the first machine. It reads source data and constructs a local data distributor. The data distributor distributes source data and combines the results into the desired output as shown in Figure 4.

[2] Although logarithmic run-time for this operation is expected on a PRAM, it cannot be achieved in practical parallel environments.

Additionally, parallel I/O is established to avoid bottlenecks within the data distributor.

The data distributor starts threads on the remote machines after distributing source data among the processors. It is important to assign each thread to a different machine to obtain a well-balanced load between the processors.

For several input planes, i.e., for an additional s-axis, optimal performance is obtained if these planes are distributed among groups of processors each computing a distinct plane. Therefore, the data space described by the t-, v- and s-axis can be divided in any way along the v- and s-axis or the t- and s-axis respectively. It is up to the distribution of the source whether separation of one of both axes or a mix might be better.

5 Benchmark Setup

We implemented parallel Veltran three times: in JavaParty, in Fortran90, and in HPF. For a specific set of input data, we took runtime measurements on both an SGI Origin2000 and an IBM SP/2. This section discusses the benchmark setup in detail.

The benchmarks are conducted on up to eight nodes of an SGI Origin2000 (shared memory parallel machine) as well as on the same number of nodes of an IBM SP/2 (distributed memory machine). See Figure 5 for the machines' technical data.

SGI Origin2000

8 195MHz IP27 Processors
CPU: MIPS R10000 Processor Revision: 2.6
FPU: MIPS R10010 Floating Point Chip Revision: 0.0
Secondary unified instruction/data cache size: 4 MB
Data Cache Size: 32 KB
Instruction Cache Size: 32 KB
Main memory size: 3072 MB
Operating System: Irix 6.4

IBM SP/2

8 66MHz RS/6000 Processors
Data Cache Size: 128 KB
Instruction Cache Size: 32 KB
Main memory size: 64 MB per node
Operating System: AIX 4.1

Figure 5: Data sheets of the system platforms used.

The benchmark consists of an iteration of Veltran with a variable number of input planes with $n_t = 1000$ points along the t-axis, $n_x = 60$ points along the x-axis and $n_v = 120$ points along the v-axis. I/O is not considered during the measurements since each system platform has different capabilities. The measured time comprises the steps of data distribution, computation, and data collection. Measurements are evaluated within the computation of one, four, and eight planes. For the problem size of one plane, four nodes are sufficient, since no further speedup can be obtained with additional nodes.

The JavaParty implementation uses JDK 1.1.4 with just-in-time compilation on the IBM SP/2. On each node, a separate JVM is started. The JVMs communicate by means of RMI. For the HPF measurements, version 2.2 of the Portland Group High Performance Fortran compiler is used. On the SGI Origin2000, we used JDK 1.1.5 with just-in-time compiler. Since this version offers native thread support, JavaParty did not use any RMI communication. SGI's standard parallelizing Fortran90 compiler is used for the Fortran measurements.

On both platforms, optimizing Fortran compilers have been used. Java's performance was not attained by compiled and optimized native code but instead relied on interpreters with just-in-time compilation features.

6 Results

On the SGI, our JavaParty implementation is slower than the equivalent Fortran90 program by a factor of about 4. On the SP/2, JavaParty faces a slowdown by 3. In part, the slowdowns are due to Java's mandatory and implicit array boundary checking.

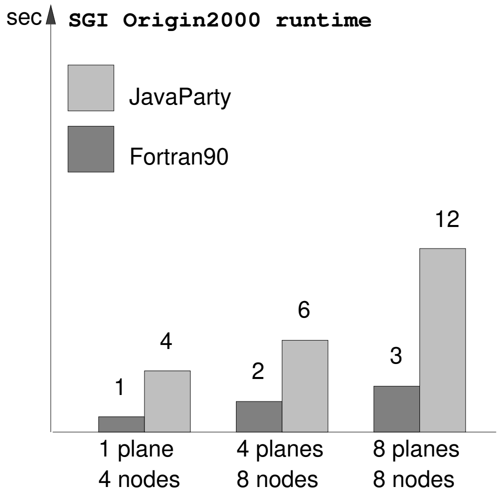

Figure 6: Measurements of parallel Veltran processing time on the SGI Origin2000.

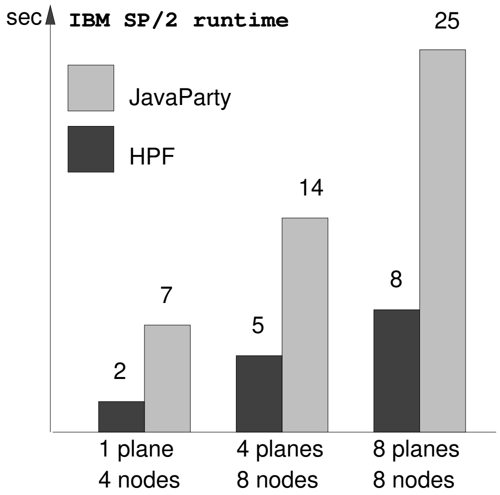

Figure 7: Measurements of parallel Veltran processing time on the IBM SP/2.

The JavaParty program automatically adapts both to the number of planes to be processed and to the number of nodes available. The Fortran programs did not have the same adaptability. Instead, we had to change some constants and recompile the Fortran code for each of the measurements. Without the manual changes and recompilation, the performance of the most general and slowest program would have shown up repeatedly. For example, to process one plane on four nodes with the general program, the Origin2000 needs 3 seconds and the SP/2 needs 8 seconds.

We expect significant performance increases for Java in the near future because of two main reasons. First, we had to use JDK 1.1.x, because later releases are not yet available for our hardware platforms. Later versions (JDK 1.2, HotSpot) have increased performance (especially RMI performance, improved native thread support and better just-in-time compilation) on Solaris and Wintel platforms and are likely to show the same effect in our environments. Second, compilers producing optimized native code like IBM's High Performance Java Compiler [7] are on the horizon.[3] These compilers will approach Fortran performance because they can apply much more sophisticated optimization techniques than current just-in-time compilers.

[3] Although the IBM SP/2 offers an alpha version of a High Performance Java Compiler [7] compiling to native and statically linked code, it is too early to seriously use that compiler. The HPJ compiler revealed a general speed up of 1.6 compared to just-in-time Java performance, but unfortunately HPJ's communication through RMI turned out to be slower by a factor of between 20 and 35.

7 Conclusion

We achieved Fortran performance within a factor of at most four with a parallel Java implementation of a basic geophysical algorithm on two major system platforms for scientific computing, both on a shared memory and a distributed memory parallel computer. Similar results can be expected for other seismic operators and on other platforms. The slowdown factor is acceptable because of three reasons: (a) object-oriented Java code is easier to maintain and reuse than Fortran code, (b) Java code is fully portable even between so different platforms, and (c) Java's performance is going to improve because of the enormous amount of research in Java base technology that is currently under way.

Acknowledgements

We would like to thank the JavaParty group, especially Matthias Zenger, for their support of the JavaParty environment. Matthias Schwab of the Stanford Exploration Project designed and provided the geophysical operator framework JAG. Furthermore we want to express gratitude to Maui High Performance Computing Center as well as Karlsruhe Computing Center for the access to the IBM SP/2 and the SGI Origin2000.

References

[1] J. Clearbout and B. Biondi. Geophysics in object-oriented numerics (GOON). In *Stanford Exploration Project Report No. 93*, pages 241–252. October 1996. http://sepwww.stanford.edu/sep.

[2] Gene Golub and Charles van Loan. *Matrix Computations*. John Hopkins, Third edition, 1996.

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[4] Siamak Hassanzadeh and Charles C. Mosher. Java: Object-Oriented programming for the cyber age. *The Leading Edge*, 15(12):1379–1381, December 1996.

[5] Siamak Hassanzadeh, Charles C. Mosher, and Calvin L. Joyner. Scalable parallel seismic processing. *The Leading Edge*, 15(12):1363–1366, December 1996.

[6] C. Holt, J. P. Singh, and J. Hennessy. Application and architectural bottlenecks in large-scale distributed shared memory machines. In *23th Int. Symp. on Computer Architecture*, pages 134–145, May 1996.

[7] IBM. High performance compiler for Java. http://www.alphaWorks.ibm.com.

[8] JavaParty. http://wwwipd.ira.uka.de/JavaParty.

[9] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability on shared virtual memory and hardware-coherent multiprocessors. In *6th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP*, pages 217–229, Las Vegas, NV, July 18–21 1997.

[10] D. Nichols, H. Urdaneta, H. I. Oh, J. Clearbout, L. Laane, M. Karrenbach, and M. Schwab. Programming geophysics in C++. In *Stanford Exploration Project Report No. 79*, pages 313–471. August 1993.

[11] Michael Philippsen and Matthias Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.

[12] Matthias Schwab and Joel Schroeder. A seismic inversion library in Java. In *Stanford Exploration Project Report No. 94*, pages 363–381. May 1997. http://sepwww.stanford.edu/sep/jag.

[13] J.R. Skewchuck. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie-Mellon University, Pittsburgh, March 1994. CS-94-125.