

A Hybrid MPI-OpenMP Implementation of the Conjugate Gradient Method in Java

H. Martin Bückler¹, Bruno Lang², Hans-Joachim Pflug³, and Andreas Wolf¹

¹ Institute for Scientific Computing, Aachen University,
D-52056 Aachen, Germany

² Applied Computer Science Group, University of Wuppertal,
D-42097 Wuppertal, Germany

³ Center for Computing and Communication, Aachen University,
D-52056 Aachen, Germany

Abstract. The message passing paradigm is the dominating programming model for parallel processing on distributed-memory architectures. OpenMP is the most widely used parallel programming model for shared-memory systems. Contemporary parallel computers consist of multiple shared-memory computers connected via a network so that a hybrid parallelization approach combining message passing and OpenMP is often desired. We present a hybrid parallelization of an iterative method for the solution of linear systems with a large sparse symmetric coefficient matrix which is implemented in Java. To this end, we bring together mpiJava and JOMP and compare the performance on a Sun Fire 6800 system with a hybrid parallel version of the same problem implemented in Fortran. The scalability of both implementations is similar while, surprisingly, the performance of the Fortran version with data prefetching is only a factor of about 2 larger than the Java version when up to 22 processors are used.

1 Introduction

Java is emerging as the de facto standard language for platform independent software development in higher education and is also becoming increasingly popular in scientific and engineering applications. Though scientific and engineering computation is certainly not its primary target area and its inefficiency has often been blamed, Java is today seriously considered for high-performance computing because its performance has been significantly increased over the last years. In a few years from now, Java might even obtain better performance than C++ or Fortran; see [3, 9, 15, 19] and the references therein for details on Java in connection to high-performance computing.

By its feature of native threads [12, 16], Java offers simple and tightly-integrated support for multiple tasks that behave as though they are all in progress at one time. Thus, Java is interesting from a parallel computing point of view. Though there is no integrated support for distributed- and shared-memory programming, preliminary prototype interfaces to the Message Passing Interface

(MPI) [14, 21] and OpenMP [17, 18] standards are available. With the emergence of multiple shared-memory computers connected via a network, a hybrid parallelization is becoming more and more attractive. In such an approach, an application is developed as an ensemble of distributed-memory processes (MPI) with shared-memory parallelism (OpenMP) used within each process [20].

As a computational problem which frequently arises from different scientific and engineering areas, we consider the iterative solution of a large sparse system of linear equations with a symmetric positive definite coefficient matrix by the conjugate gradient method [8]. The aim of this note is different from the work in [7, 11, 13] where, using Fortran, different hybrid parallelization strategies for this given problem at hand are described. Here, the focus is on the programming language Java and its connection to parallel programming models which are vital in large-scale computational science. In particular, we demonstrate that Java's preliminary prototype interfaces to the MPI and OpenMP standards work together. To the best of our knowledge, this is the first time a hybrid parallelization approach is used for Java programs.

The structure of this note is as follows. In Sec. 2, references to Java's prototype interfaces to the MPI and OpenMP standards are given. A simple MPI-OpenMP approach to parallelize the conjugate gradient method is sketched in Sec. 3. Experimental results of the hybrid Java implementation are reported in Sec. 4. The Java implementation is also contrasted with a hybrid reference implementation in Fortran.

2 Parallel Computing in Java Using mpiJava and JOMP

MPI is the dominating programming model for parallel processing on distributed-memory architectures. OpenMP is the most widely used parallel programming model for shared-memory systems. For Java, prototype interfaces to the MPI and OpenMP standards are available.

MPI [14, 21] is a specification for a set of portable functions for managing communication between multiple processes. The standard is available via <http://www.mpi-forum.org>. Official bindings are defined for C/C++ and Fortran. The Java Grande Forum [9] proposed a reference message passing interface for Java called MPJ [1]. Unfortunately, there is currently no complete implementation of MPJ. However, the mpiJava [2] package is currently implemented as an object-oriented Java interface to an underlying MPI implementation. Version 1.2 of mpiJava is close to MPJ.

OpenMP [5, 17, 18] is a set of portable compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C/C++ and Fortran programs. The standard defined by a group of computer hardware and software vendors is available via <http://www.openmp.org>. The University of Edinburgh developed a package called JOMP [4, 10] providing an OpenMP-like set of directives and library routines for Java heavily based on the existing OpenMP C/C++ specification and enables a high-level approach to

shared-memory programming. A prototype compiler and a runtime library are available.

3 Hybrid Parallelization Approach for the CG Method

In order to demonstrate the feasibility of a hybrid parallelization strategy and to assess its quality, we consider the iterative solution of a large linear system of equations with a sparse coefficient matrix. To this end, we solve the two-dimensional Laplace equation on the unit square. Finite differences with a five-point stencil on a regular grid are employed to discretize the partial differential equation leading to a symmetric coefficient matrix. The conjugate gradient method (CG) [8] is applied to iteratively solve the resulting sparse linear system. For the sake of simplicity, preconditioning which is crucial in solving linear problems from real-world applications is not considered throughout this note.

The conjugate gradient method consists of a loop over a fixed number of matrix-vector and vector-vector operations. More precisely, the basic operations of a single CG iteration involve the product of the coefficient matrix and a vector as well as some linear combinations and inner products of vectors. For the MPI processes, a two-dimensional domain decomposition approach is used to parallelize the computation and to distribute the corresponding data. All vectors are distributed consistently over the MPI processes so that no communication is needed in vector updates. Within each MPI process, OpenMP is used to further parallelize vector updates, matrix-vector products, and inner products. Since we are interested in the feasibility of a hybrid approach in Java, we do not try to minimize synchronization. The aim is not to develop the most efficient parallelization strategy but rather to concentrate on the general procedure of applying a hybrid parallelization in an incremental manner. Therefore, we choose a simple way of OpenMP parallelization by using a combined parallel work-sharing construct for each computation, namely a *parallel do* directive with appropriate clauses.

There are two routines needing MPI communication as well as OpenMP directives. The first one is the matrix-vector product, where the MPI parallelization is implemented by exchanging the data at the boundaries of each two-dimensional subgrid of an MPI process. This is done still outside of any OpenMP parallel region. After the MPI communication has completed, each process opens an OpenMP parallel region and calculates its results local on each MPI process. The second one is the inner product. Here, a global communication over the MPI processes will be applied after the processes have calculated their local sums in an OpenMP parallel region.

4 Experimental Results

In the experiments reported in this section, a SUN Fire 6800 system with 24 Ultra Sparc III processors (900MHz) and 24 GByte of main memory is used. All experiments are carried out with Solaris 8. The hybrid Fortran program is

aggressively compiled with Forte[™] Developer 7.1 using MPI from version 5 of the SUN HPC cluster tools. The hybrid Java program makes use of version 1.2.3 of the mpiJava package and JOMP, version 1.0.beta. The Java(™) 2 Runtime Environment, version 1.4.1 is employed. The SUN Fire 6800 system supports data prefetching as a crucial ingredient for improving the performance of an application. Data prefetching refers to the preloading of data into the cache to prevent cache misses [6]. Data prefetching can be switched off for Java by changing the hardware settings and for Fortran by disabling the corresponding compiler option.

When increasing the number of processors during the experiments, we simultaneously increase the order of the linear system in the following way. The number of vector elements per processor is always about five times the number of vector elements that would fit into cache. In this way, the influence of cache effects is minimized. Furthermore, the goal is to solve problems that are large enough so that the overall execution time is dominated by the computational work and not by communication/synchronization. Also, this procedure allows us to present performance results of the same hybrid code throughout the remainder of this note. We stress that the differences in the performance between the hybrid code and two “pure” versions, one using only MPI and another using exclusively OpenMP, are negligible and give no indication for any performance loss. Finally, we mention that the single-processor performance of the hybrid code is almost identical to a corresponding serial implementation.

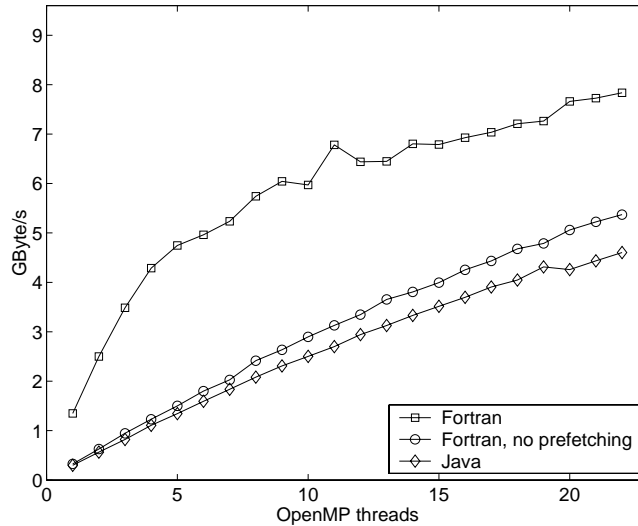


Fig. 1. Memory bandwidth usage in GByte/s for OpenMP distribution of the hybrid implementation.

As a first performance metric we consider the memory bandwidth of the application which cannot exceed the theoretical hardware limit of 9.6 GByte/s. In Fig. 1, the memory bandwidth of the hybrid code is shown where a single MPI process, i.e., OpenMP-only parallelization, is used. The graphs give the memory bandwidth versus the number of OpenMP threads for three different configurations: a Fortran version with data prefetching labeled “Fortran,” a Fortran version without data prefetching labeled “Fortran, no prefetching,” and a Java implementation. It turns out that, in Java, there is no performance difference between with and without data prefetching. The Fortran performance decreases by a factor of 2 to 3 when disabling data prefetching. The Fortran version without data prefetching performs only slightly better than the Java version. This leads us to the conclusion, that the lack of data prefetching support is the main reason for the performance loss in Java. Thus, better data prefetching support should bring the Java version closer to Fortran speed. This can be implemented e.g. by special support for vector operations.

Another observation from this figure is that the memory bandwidth of the Fortran program with data prefetching saturates with increasing number of threads and tends to reach the memory bandwidth limitation. The performance of this configuration gets more and more bandwidth limited until it reaches a bandwidth of about 8 GByte/s on 22 processors. The Java implementation as well as the Fortran implementation without data prefetching don't reach the bandwidth limitation and scale well up to 22 processors.

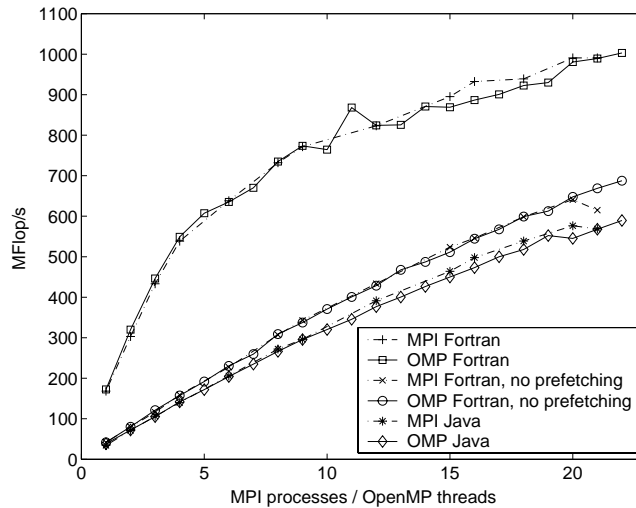


Fig. 2. Performance of the hybrid implementations distributing the work either by MPI or OpenMP.

In the next set of experiments, the work of the hybrid code is not only distributed by OpenMP but also by MPI. However, we first consider the case where OpenMP and MPI are used separately. The performance results for the Java and Fortran programs distributing the work by either MPI or OpenMP are given in Fig. 2. The single-processor performance of the hybrid Java program, 40 MFlop/s, is near 23% of the performance of the Fortran program with data prefetching, 177 MFlop/s. As compared to this Fortran implementation, the lower performance of the floating point operations in Java results in a smaller synchronization influence and leads to a better scaling behavior for the Java program. Notice that the performance gap between Java and Fortran with data prefetching is decreasing with an increasing number of processors. For instance, with 10 processors, the Java program speed increases to 42% of the Fortran reference, and for 22 processors the percentage is about 57%. The Fortran implementation without data prefetching and the Java code are comparable.

In the last set of experiments, the hybrid code is run with using OpenMP and MPI simultaneously. In Figs. 3 and 4, the performance results are given when the work is distributed by MPI *and* OpenMP. In these figures, a vertical bar is used to denote the performance values of the Java and Fortran implementations where the Java performance is indicated by the height of the smaller bar and the Fortran performance is represented by the height of the overall bar. The Java and Fortran (without data prefetching) versions both scale well when increasing the number of MPI processes or the number of OpenMP threads; see

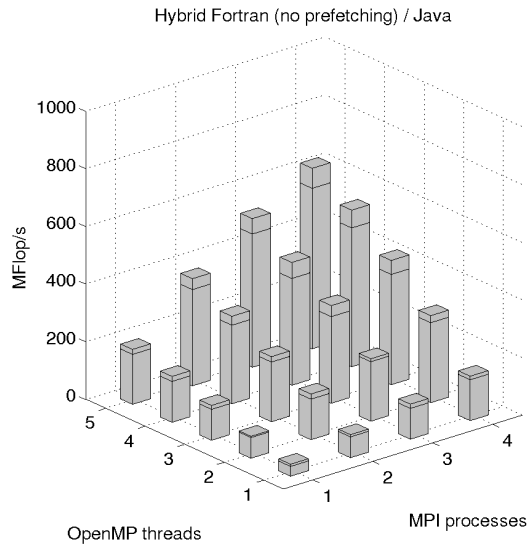


Fig. 3. Performance of the hybrid Fortran without data prefetching (higher values) and Java (lower values) implementations.

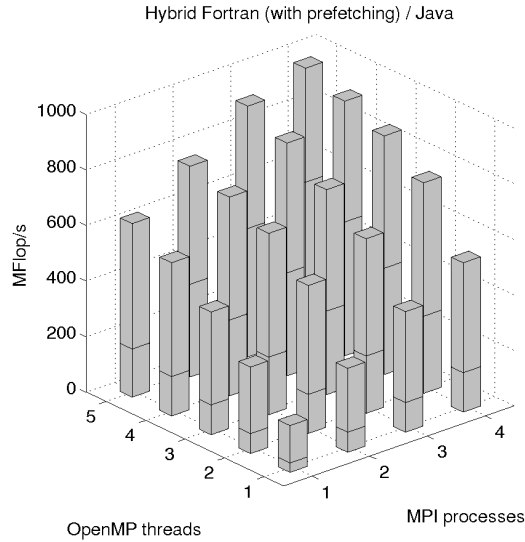


Fig. 4. Performance of the hybrid Fortran with data prefetching (higher values) and Java (lower values) implementations.

Fig. 3. The performance figures obtained with hybrid parallelization are close to those reported in Fig. 2 where the work is distributed exclusively by either MPI or OpenMP. Note that, for 20 processors, the performance of the Java program, 584 MFlop/s, is about 58% of the performance of the Fortran (with data prefetching) program, 1014 MFlop/s; see Fig. 4.

5 Concluding Remarks

In the area of high-performance computing, Java has often been blamed for its inefficiency. However, a lot of effort was put into the development of compilers and virtual machines to enhance the performance of Java. The performance gap between Java and Fortran has continually decreased in the last few years. The rapid development time in coding, the option to integrate existing applications written in other programming languages, the standardized infrastructure for distributed applications in a potentially heterogeneous environment, and the well-integrated graphical interface for data visualization make Java an interesting candidate for high-performance computing.

In the iterative solution of a large sparse linear system using the conjugate gradient method, the hybrid MPI-OpenMP parallelization in Java scales well and the performance gap between Java and a corresponding hybrid Fortran program is decreasing when the number of processors is increased. On the SUN Fire 6800 system used throughout this study, its data prefetching mechanism is identified

as the main reason for the loss of the performance in Java. Nevertheless, the use of Java in high-performance computing offers an interesting and promising research area for the not too far future.

Acknowledgments

The authors would like to thank Dieter an Mey and Andrea Lorenz of the Center for Computing and Communication at Aachen University, Germany, for their cooperativeness during this work.

References

- [1] M. Baker and B. Carpenter. MPJ: A proposed Java message passing API and environment for high performance computing. In J. Rolim et al., editors, *Parallel and Distributed Processing: Proceedings of the 15 IPDPS 2000 Workshops, Cancun, Mexico, May 2000*, volume 1800 of *Lecture Notes in Computer Science*, pages 552–559, Berlin, 2000. Springer.
- [2] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lang. mpiJava: An object-oriented Java interface to MPI. In J. Rolim et al., editors, *Parallel and Distributed Processing: Proceedings of the 11 IPPS/SPDP'99 Workshops, San Juan, Puerto Rico, USA, April 1999*, volume 1586 of *Lecture Notes in Computer Science*, pages 748–762, Berlin, 1999. Springer.
- [3] R. F. Boisvert, J. Moreira, M. Philippsen, and R. Pozo. Java and numerical computing. *IEEE Computing in Science and Engineering*, 3(2):18–24, 2001.
- [4] J. M. Bull and M. E. Kambites. JOMP – an OpenMP-like interface for Java. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 44–53, 2000.
- [5] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufman Publishers, San Francisco, CA, 2001.
- [6] R. P. Garg and I. Sharapov. *Techniques For Optimizing Applications: High Performance Computing*. Prentice Hall / Sun Microsystems Press, Palo Alto, CA, 2002.
- [7] L. Giraud. Combining shared and distributed memory programming models on clusters of symmetric multiprocessors: Some basic promising experiments. *The International Journal of High Performance Computing Applications*, 16(4):425–430, 2002.
- [8] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.
- [9] Java Grande Forum. <http://www.javagrande.org/>.
- [10] M. E. Kambites, J. Obdrzalek, and J. M. Bull. An OpenMP-like interface for parallel programming in Java. *Concurrency and Computation: Practice and Experience*, 13(8–9):793–814, 2001.
- [11] P. Lanucara and S. Rovida. Conjugate-gradients algorithms: An MPI–OpenMP implementation on distributed shared memory systems. In *Proceedings of the First European Workshop on OpenMP – EWOMP'99, Lund, Sweden, September 30 – October 1, 1999*, pages 76–80, 1999.

- [12] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, 1996.
- [13] G. Mahinthakumar and F. Saied. A hybrid MPI-OpenMP implementation of an implicit finite-element code on parallel architectures. *The International Journal of High Performance Computing Applications*, 16(4):371–391, 2002.
- [14] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4):159–416, 1994.
- [15] J. E. Moreira, S. P. Midkiff, M. Gupta, P. Wu, G. Almasi, and P. Artigas. NINJA: Java for high performance numerical computing. *Scientific Programming*, 10(1):19–33, 2002.
- [16] S. Oaks and H. Wong. *Java Threads*. O’Reilly, 1997.
- [17] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 1.0, 1998.
- [18] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0, 1999.
- [19] M. Philippsen, R. F. Boisvert, V. S. Getov, R. Pozo, J. Moreira, D. Gannon, and G. Fox. JavaGrande – Work and Results of the JavaGrande Forum. In T. Sørevik, F. Manne, R. Moe, and A. H. Gebremedhin, editors, *Applied Parallel Computing: New Paradigms for HPC in Industry and Academia, Proceedings of the 5th International Workshop, PARA 2000, Bergen, Norway, June 2000*, volume 1947 of *Lecture Notes in Computer Science*, pages 20–36, Berlin, 2001. Springer.
- [20] L. Smith and M. Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2–3):83–98, 2001.
- [21] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, MA, 2nd edition, 1998.