

# Programming Models for Parallel Java Applications

Mark Bull and Scott Telford  
Edinburgh Parallel Computing Centre, Edinburgh, EH9 3JZ

## 1 Introduction

Java has yet to make a significant impact in the field of traditional scientific computing. However, there are a number of reasons why it may do so in the near future. Perhaps the most obvious benefits are those of portability and ease of software engineering. The former will be particularly important when grid computing comes of age, as a user may not know when they submit their job, what architecture it will run on. Automatic garbage collection, thorough type checking and the absence of pointers make Java development significantly less error prone than more traditional languages such as C, C++ and Fortran.

Of course, using Java is not without its problems. Perhaps the prime concern for scientific users is performance, though just-in-time compilers are making rapid advances in this field. Other issues include lack of support for complex numbers and multi-dimensional arrays. For a more detailed discussion of these and other issues related to the use of Java for high end scientific applications, see [2]. A lack of suitable standards for parallel programming is also a concern: this report discusses the current status of parallel programming models for Java.

The remainder of this document is organised as follows: Section 2 draws the distinction between single and multiple virtual machine environments. Sections 3 and 4 discuss the various programming models specific to the two types of environment in turn, while Section 5 considers data parallel models which are applicable to either type. Section 6 introduces a small demonstrator application, which has been implemented in several different parallel programming models, and presents some performance results. The source codes for the different versions are included as Appendices. Finally, Section 7 summarises.

## 2 Single vs. Multiple Java Virtual Machines

The Java Virtual Machine (VM) is the program that interprets Java bytecodes. The VM is what makes Java portable—a vendor writes a Java VM for their operating system, and any Java program can run on that VM. Most VM include a just-in-time (JIT) compiler which will compile heavily used parts of the application into machine code at runtime. These parts will then typically run much faster than if they were simply interpreted.

One fundamental characteristic of any Java-based parallel system is the distinction between *single virtual machine* environments (henceforth referred to as single-VM environments) and *multiple virtual machine* (multi-VM) environments. A single-VM environment, as the name suggests, pro-

vides a single Java virtual machine, running a single multi-threaded Java application, with the Java threads distributed across the processors in the system.

A multi-VM environment provides multiple Java VMs, each running a discrete Java application, with some form of inter-VM communication facility allowing interaction between the applications running on different VMs. Each VM in a multi-VM environment may be running on either a single processor or an SMP multiprocessor system and may be running a multi-threaded application. In the most general case, therefore, a multi-VM system can be thought of as a collection of multi-threaded Java applications communicating (outside the Java thread model) via some unspecified mechanism.

A single-VM environment is naturally suited to a hardware architecture which provides a *single system image*, i.e. one which presents a single address space to applications, irrespective of the physical distribution or proximity of memory and processors. Such a hardware architecture enables the use of existing Java VM software implementations with little or no modification. In addition, the widely-used and understood Java threads model can be employed in the application and no extra APIs or non-standard changes to the code are required, assuming there is a sufficient degree of threading to keep all the processors reasonably busy. However, such platforms tend to be difficult to scale past a few hundred processors, both in terms of hardware and applications.

For a hardware architecture without a single system image, e.g. a classic MPP supercomputer or “Beowulf” cluster, a multi-VM environment is the obvious choice. However, research projects at the IBM Research Laboratory in Haifa [1] and Rice University, Texas [11] have investigated the design and implementation of Java VMs capable of providing a single-VM environment across such a distributed-memory architecture. This is a complex and ambitious approach and the efficiency of such a VM is uncertain and likely to be highly variable, depending on the behaviour of the running Java application.

A multi-VM environment would map Java VMs to separate address spaces (e.g. individual nodes in a Beowulf cluster). This requires a Java API for the inter-VM communication which hopefully maps efficiently to the processor interconnect hardware of the chosen platform. As mentioned above, each address space may possibly correspond to multiple processors, thus allowing the use of multi-threaded applications.

Again, little or no changes would be required to existing Java VMs, with the possible exception of the addition of an efficient native interface to the interprocessor communication mechanism. In the general case, this gives a two-level parallelism model (threads plus inter-VM communication). While this results in more complex applications software, a multi-VM environment can exploit the greater scalability of non-single system image hardware platforms. Moreover, for reasonably large numbers of processors, non-single system image platforms need much less sophisticated hardware and are hence simpler and cheaper to build.

In summary, there is a trade-off between potential scalability and ease of programming: using the thread model native to the Java programming model will probably limit the scalability of the application. However, the use of another (possibly non-standard) communications model in addition to or instead of threads, as required by a multi-VM environment, may prove awkward.

## 3 Single-VM Programming Models

### 3.1 Java Threads

For single-VM Java environments, as defined in Section 2, it is natural and appropriate to use Java threads as the parallel programming paradigm, since the thread class is part of the standard Java libraries. Unlike earlier Java implementations, most current Java VMs implement the thread class

on top of the native OS threads (for example, Sun's JVM for Solaris uses Solaris threads). This means that the threads are visible to the operating system and are capable of being scheduled on different processors, allowing useful parallel execution.

However, writing parallel programs using Java threads requires a somewhat different approach from writing threaded programs for concurrent execution on a single processor, which is the more traditional use for the thread class. Firstly, it is important to avoid thread context switching by ensuring that the number of threads matches the number of processors used. Instead of creating a new thread to run every object capable of parallel execution, it is necessary to create and manage a task pool, where a fixed number of threads execute tasks assigned to them. In principle, this functionality could be implemented in the thread scheduling of a multi-processor aware Java VM, but for present VMs, this has to be done in user code. Creating and killing threads is expensive, so the threads must stay alive for the duration of the parallel program. A further feature necessary for good parallel performance is a fast barrier synchronisation method—no such method is provided by the thread class, and significant care must be taken to design a scalable solution.

### **3.2 OpenMP**

Much of the parallelism in scientific applications occurs at the loop level. The most obvious way to exploit parallelism using Java threads is at the method level, which is often not sufficiently scalable. To execute a loop in parallel using Java threads, it is necessary to create a new method containing the loop, compute appropriate loop bounds for each thread, and pass an instance of the method's class to the task pool manager for execution on each thread. This is a somewhat clumsy approach, which is not unique to Java threads — the same issues are involved in writing shared memory parallel programs using Fortran or C and, for example, the POSIX threads library. However, much of this process can be readily automated using compiler directives.

In the past, compiler directives for shared memory parallel programming were usually vendor specific and suffered from a lack of standardisation. However, OpenMP, an open standard for shared memory directives has recently been published [13, 14]. OpenMP has been widely adopted in the industry and is now supported by most of the major vendors, including Cray, Compaq, IBM, Sun and SGI. Currently, OpenMP defines directives for Fortran, and C/C++ only. Recent work at EPCC has also defined and implemented a prototype Java version of OpenMP called JOMP [6], using Java threads as the underlying parallel model. This raises the possibility of a straightforward and familiar route to using Java for shared memory numerical applications, and the porting to Java of existing Fortran, C, or C++ applications using OpenMP whilst keeping the same parallelism model.

### **3.3 Auto-parallelisation**

Much research has been directed towards the goal of completely automating the task of porting sequential codes to shared memory parallel architectures using threads libraries. This has proven to be a very difficult task, even when restricted to loop parallelisation, which involves both analysing data dependencies and using loop transformation strategies in order to remove them where possible. Most of this work has been directed at Fortran, and more recently C, but much of the technology can carry over to Java, as illustrated by the prototype compiler javar [5].

Auto-parallelisation has not, however, gained wide acceptance amongst applications developers. The compiler must be conservative in cases where dependency analysis indicates possible race conditions, and can therefore fail to parallelise loops which the programmer knows are in fact safe to execute in parallel. The data dependency analysis can struggle to cope with large loop bodies and loops which contain calls to other procedures. Furthermore, the transformations applied are essentially local to each loop, since global optimisation strategies to optimise for data locality are

extremely difficult to design.

## 4 Multi-VM Programming Models

A multi-VM environment requires some mechanism for inter-VM communication. The three obvious approaches to providing this are:

- To use or adapt a suitable existing Java API. The only current official Java API which is potentially suitable for this purpose is Java RMI.
- To adopt an existing, widely-used, non-Java native inter-processor communication library and provide a Java interface to it. Examples of this are Java interfaces to MPI and VIA (see Section 4.3).
- To extend the Java programming language to add intrinsic parallelism features. An example of this is JavaParty.

All of these approaches are described more fully below.

### 4.1 Java RMI

Java RMI (*Remote Method Invocation*) [16] is an API which supports seamless remote method invocation on objects in different Java VMs on a network. This allows the implementation of distributed applications using the Java object model.

RMI is primarily intended for use in a client-server paradigm: A server application creates a number of objects; a client application then invokes methods on these objects remotely. This paradigm is not commonly used in high performance scientific applications (except perhaps in cases where there is trivial task parallelism). Most decomposition techniques require the passing of data between processes, which is then processed locally, rather than invoking computation on remote data. While it is possible to write new applications from scratch in this paradigm, porting existing applications is likely to be awkward.

Furthermore, RMI suffers from significant performance penalties, resulting in latencies which are likely to prove unacceptable in a high performance context. The internal design of Java RMI is intended to have an abstract protocol-independent underlying transport layer with protocol-specific code in subpackages thus easing the implementation of alternative transports. As of the Java 2 SDK release 1.2 RMI, only a TCP-based transport is implemented. However, provision for the use of other socket-based protocols is included in the RMI API.

Due to this dependence on TCP/IP, the current RMI implementation is unsuitable for use in a closely-coupled parallel system. However, in theory, the internal structure of RMI should allow alternative underlying transport layers based on more appropriate interfaces such as MPI or VIA to be implemented. EPCC has recently investigated the feasibility of porting RMI to an MPI-based transport using mpiJava [17]. Unfortunately, in the 1.2 release RMI source code, there are many TCP/IP-centric assumptions made in the allegedly “transport-independent” packages which make such an adaptation infeasible. From communication with the RMI developers, it is understood that future releases of RMI will have these assumptions removed. This should make alternative RMI transport implementations more feasible. However, the performance of Sun RMI is still likely to be disappointing, even with a high-performance transport, as much of the overhead is in the software layers.

Both RMI and Java object serialization (the process required to transform an object, and, recursively, all the objects it refers to, into a byte stream suitable for communication) as currently implemented by Sun are significantly computationally expensive. Alternative third-party implementations of both, such as the University of Karlsruhe's KaRMI [12] and UKA-Serialization have shown considerable improvements in efficiency, but the use of RMI is always likely to incur significant extra costs compared to using the underlying transport mechanism more directly. Using RMI simply to pass data between multiple VMs also requires the creation of additional threads to manage remote requests for data, incurring context switching and synchronisation overheads. The implicitly blocking nature of remote methods make latency hiding techniques difficult to implement. It therefore seems unlikely that RMI will prove a popular choice for parallel applications.

It is worth noting here some other work carried out at University of Karlsruhe to provide a higher level interface to distributed/parallel applications using RMI called JavaParty [15]. This consists of an extension to the Java language (a remote qualifier for objects), a compiler to translate the augmented language to Java with RMI calls, and a run-time system which manages the distribution of the remote objects. Some useful speedups are reported on a geophysics application using this system.

## 4.2 Message Passing

In high-performance computing, the most widely-used standard for inter-process communication is the *Message Passing Interface* or MPI [10]. This is an open standard for a message-passing library which defines bindings for C and Fortran 77, the later MPI-2 standard also defining C++ and Fortran 90 bindings.

Due to its origins as a C/Fortran library, the design of MPI is distinctly non-object-oriented and somewhat alien to the Java environment. However, there have been several research efforts to provide MPI for Java. Due to the reluctance of the MPI Forum to become involved in defining further language bindings for MPI, these efforts are now focussed within the Message Passing Working Group of the Java Grande Forum. A draft specification for a standard Java API for message passing has been published [7] and this is now evolving towards a more Java-centric design, and away from strict MPI conformance. This work, now referred to as MPJ, is still in progress and no further specifications have been released, although some discussion documents have been written [3].

It is worth noting that MPI for Java can (and has) been implemented in two quite different ways: as a Java wrapper around a native MPI library; or entirely in Java. However, to implement entirely in Java requires the use of a core Java API as the underlying communications layer, such as `java.net`, which in turn implies the use of TCP/IP. Hence, for a closely-coupled parallel system with some fast processor interconnect, only a native MPI implementation is appropriate, for performance and architectural reasons. An example of the "native library wrapper" approach is NPAC's `mpiJava` [4], which uses the MPI C++ binding model as a basis and interfaces to several different native MPI implementations. A future possibility would be the implementation of MPI in Java using some non-standard Java API, such as VIA (see below).

## 4.3 VI Architecture

The *VI (Virtual Interface) Architecture* [9] is a specification devised by Compaq, Intel and Microsoft for a high-performance C API that gives user level processes direct but protected access to network interfaces. This is intended for system area networks (i.e. server clustering) but is also applicable to HPC clusters.

The VIA specification can be implemented to varying degrees in hardware — a largely software implementation can utilise standard Fast Ethernet or Gigabit Ethernet network interface hardware. Alternatively a specially-designed VIA hardware interface can implement most of the specification in hardware. VIA is of necessity a low-level interface, and in an HPC context, would be likely to serve as an underlying communications layer for another API, such as MPI.

VIA virtualises a physical hardware interface (hence the name), allowing direct control of the interface by a user application, without compromising system security or robustness. A key feature of VIA is the use of “pinned” buffer memory accessible to both (virtual) network interface and user process, thus avoiding buffer-copying overheads.

At least two current research projects aim to provide a Java interface to VIA: Javia [8] at Cornell University and Jaguar [19] at the University of California at Berkeley. Obviously, such a low-level mechanism requires some changes to the behaviour of the Java native method interface.

Javia extends the JNI native method interface with a hybrid Java/native non-garbage-collected buffer datatype which can be used as “pinned” VIA buffers. Jaguar takes a more radical approach to the same problem, by extending Java bytecode with special low-level system resource access instructions, which are then translated into native code routines which are able to access the VIA buffers. Currently, this translation is only implemented in an extension to a native-code Java compiler (Cygnum’s gcj), although it could also be implemented as part of a JIT compiler in a Java VM.

## 5 Data-parallel programming

In numerical scientific applications, a common method of parallelisation is *data parallel* programming. The idea behind the data parallel programming paradigm is to distribute arrays across multiple processors. Computation is then scheduled on the processor owning the relevant array elements.

Usually, languages such as C or Fortran are used, with the data parallel operations denoted by means of directives in the code which are used by a pre-processor to generate the code required to implement the data distribution. An alternative approach is taken by HPF (High Performance Fortran), which integrates the data-parallel directives into the core language and compiler. The code generated by the pre-processor or compiler will vary according to the target platform, but may utilise MPI [10] to implement the data-parallelism. Implementation on shared memory systems is of course possible, using the native threads library.

There are two research efforts of note which apply this concept to Java, providing language extensions that support data parallel programming. Titanium [21] from University of California at Berkeley, is a dialect of Java designed for large-scale scientific computing. The main additions to Java are immutable classes, multi-dimensional arrays, an explicitly parallel SPMD model, and zone-based memory management. Spar [18] from TU Delft also extends Java, with explicitly parallel loops, templates, tuples, and multidimensional arrays. It also targets heterogeneous systems, and could be used, for example, to program a digital signal processor (DSP) array together with a control processor.

However, the prototype compilers for both these language extensions do not emit plain Java source or byte code (instead they compile to C or native machine code), so their output cannot be run by a JVM.

Nevertheless, these language extensions, or something similar, could in principle be used for parallel programming in both single and multi-VM environments.

## 6 Demonstrator Application

This section describes versions of a very simple code which illustrates the use of various parallel programming models in Java.

The application is a simulation of two dimensional fluid flow in a box with inflow and outflow through apertures in adjacent faces of the box. The 2-D steady-state viscous incompressible Navier-Stokes equations in the usual streamfunction/vorticity formulation are solved on a regular grid using a red-black Gauss-Seidel relaxation method on the classical 5-point stencil.

This application was chosen as it is simple and the implementation is compact, but it is reasonably representative of explicit finite-difference codes which are common in scientific applications.

### 6.1 Sequential Implementation

The sequential Java implementation consists of a single class `Inject`. The main method creates a new instance of an `Inject` object which contains the various geometrical and numerical parameters together with the main field arrays. The solve method is then called on this object, which implements the Gauss-Seidel relaxation. Two additional methods `boundarypsi` and `boundaryzet` are used to implement the relevant boundary conditions. The source code for this version is in Appendix A.

### 6.2 Parallel Implementations

#### 6.2.1 Java threads

The Java threads version (see Appendix B) introduces a new class, `gridrunner`, which implements the `Runnable` interface. The relaxation code has been moved to this class's `run` method. The solve method of `Inject` now creates the requested number of `Thread` objects, and passes a new `gridrunner` object to each thread, which then calls its `run` method. Note also the addition of the `Barrier` class which provides fast global synchronisation between the threads.

In this implementation the boundary condition code has not been altered. The setting of boundary conditions is executed sequentially by thread 0.

#### 6.2.2 JOMP

The JOMP version of the code (Appendix C) shows the equivalent code to the Java threads version as it would appear in the suggested OpenMP-like API. The code is identical to the sequential version with the addition of two `parallel for` directives which indicate that the two main loops in solve should be executed in parallel. The JOMP compiler is run on this version to produce a plain Java source file with calls to the JOMP runtime class library. This code can then be compiled and executed as a normal Java threads code, with the number of threads being determined at runtime via a Java system property flag.

#### 6.2.3 mpiJava

In the `mpiJava` version (Appendix D), each process runs its own copy of the code, so the `Inject` object contains a subset grid. A standard one-dimensional decomposition is used, with each process holding its own subset of rows plus copies of the two neighbouring rows (the halo region).

The main method contains calls initialise and finalize MPI, as well as determining the number of processes and the rank of the calling process. Most of the communication takes place in a new method, haloexchange, which swaps copies of grid rows between neighbouring processes. There is also a call to a global reduction to compute the residual. Note that a buffer array is used here—the interface does not allow primitive data types to be used in MPI operations.

### 6.3 Performance

The three parallel Java versions of this code were run on a Sun HPC 6500 SMP system with 18 400MHz processors, each having 8Mb of Level 2 cache. The JVM used was Sun’s Solaris production JDK, Version 1.2.1\_04, and the mpiJava version used MPICH Version 1.1.2. For comparison, a Fortran version of the code (using OpenMP directives and compiled with the KAI guidef90 compiler, Version 3.7) was also tested. The source code for this version is in Appendix E. Table 1 shows the execution time and performance for 100 red-black iterations on a  $1000 \times 1000$  grid.

No. of procs	Fortran/OpenMP		Java/MPI		Java/Threads		Java/OpenMP	
	Time	Perf	Time	Perf	Time	Perf	Time	Perf
1	47.5	92	97.2	45	97.9	44	98.4	44
2	24.2	181	38.2	115	48.7	90	48.7	90
4	12.6	348	19.3	227	24.8	176	24.9	176
8	6.4	684	10.3	424	12.1	362	12.3	356
12	4.3	1010	9.0	486	7.8	558	8.2	536
16	3.2	1360	8.3	528	5.8	752	6.1	716

Table 1: Execution time (in seconds) and performance (in Mflop/s) of demonstrator application on Sun HPC 6500

Figure 1 shows the performance of the versions of the codes compared to ideal speedup calculated from the performance of sequential Fortran and Java versions. Of the Java versions, the

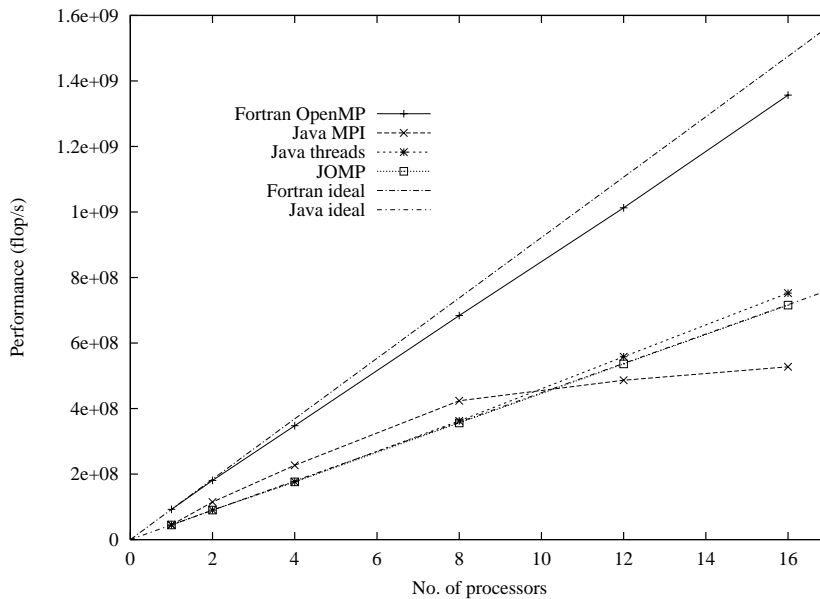


Figure 1: Performance of demonstrator application on Sun HPC 6500

threads version gives the best scalability (slightly superlinear), and is only about a factor of two slower than the Fortran/OpenMP version. The mpiJava version shows some superlinearity on small numbers of processors (presumably due to cache effects) but its performance tails off rapidly beyond eight processors. The JOMP version exhibits only slightly lower performance than the hand coded threaded version. It is worth noting that the JOMP version requires significantly less programmer effort than either the threads or MPI versions.

## 7 Summary

For single-VM systems, Java's threads class is entirely suitable as a parallel programming model, provided some additional utility classes to provide efficient scheduling and synchronisation are available. An OpenMP-like extension to Java is feasible, and would provide an interface which is both neater and more familiar to scientific programmers. Auto-parallelisation technology can also be applied to single-VM Java environments, but there is no reason to believe that it will prove any more successful than it has for Fortran or C.

For multiple-VM systems the situation is less clear. Java's distributed communication mechanism, RMI, is clearly the wrong model for parallel computing (no-one in the HPC community uses RPC to do parallel programming in C!) and also suffers from serious efficiency problems. Although performance can be improved by better implementation, this is not likely to be sufficient; if one wishes to use a fast network, latencies will be software dominated.

A message passing interface for Java would be a much better solution for parallel programming in a multiple-VM environment. Although interfaces to native MPI libraries are available, standards are lacking and a more object oriented solution such as that currently being considered by the Java Grande Forum would be preferable. Data parallel extensions to Java are also possible, but these are currently very much in the research phase.

## References

- [1] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proc. IEEE International Conference on Parallel Processing 1999 (ICPP-99)*. IEEE, September 1999. See also [www.haifa.il.ibm.com/projects/systech/cjvm.html](http://www.haifa.il.ibm.com/projects/systech/cjvm.html)
- [2] Mike Ashworth. The Potential of Java for High Performance Applications. In *Proceedings of the First International Conference on the Practical Application of Java*, pp. 19-30, 1999. Available from [www.dl.ac.uk/TCSC/CompEng/Ashworth\\_M/pubs.html](http://www.dl.ac.uk/TCSC/CompEng/Ashworth_M/pubs.html)
- [3] Mark Baker and Bryan Carpenter. Thoughts on the structure of an MPJ reference implementation. NPAC at Syracuse University, October 1999. Available from [www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html](http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html)
- [4] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpiJava: An Object-Oriented Java interface to MPI. In *Proc. International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999*, April 1999. Available from [www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html](http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html)
- [5] Aart J C Bik, Juan E Villacis, and Dennis B Gannon. javar: A Prototype Java Restructuring Compiler. *Concurrency: Practice and Experience*, 9(11):1181-1191, 1997. See also [www.extreme.indiana.edu/~ajcbik/JAVAR](http://www.extreme.indiana.edu/~ajcbik/JAVAR)
- [6] Mark Bull and Mark Kambites. JOMP—An OpenMP-like interface for Java. In *Proc. ACM 2000 Java Grande Conference*. ACM, June 2000. See also [www.epcc.ed.ac.uk/research/jomp](http://www.epcc.ed.ac.uk/research/jomp)

- [7] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-3, Java Grande Forum, November 1998.
- [8] Chi-Chao Chang and Thorsten von Eicken. Interfacing Java to the Virtual Interface Architecture. In *Proc. ACM 1999 Java Grande Conference*. ACM, June 1999.
- [9] Compaq Computer Corp., Intel Corp., and Microsoft Corp. Virtual Interface Architecture Specification. Technical report, Compaq Computer Corp., Intel Corp., Microsoft Corp., December 1997. Draft revision 1.0. See also [www.viarch.org/default.htm](http://www.viarch.org/default.htm)
- [10] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard*. University of Tennessee, Knoxville, version 1.1 edition, June 1995.
- [11] Y Charlie Hu, Weimin Yu, Alan L Cox, Dan S Wallach, and Willy Zwaenepoel. Run-Time Support for Distributed Sharing in Typed Languages. Dept. of Computer Science, Rice University, November 1999.  
See also [www.cs.rice.edu/~willy/TreadMarks/overview.html](http://www.cs.rice.edu/~willy/TreadMarks/overview.html)
- [12] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A More Efficient RMI for Java. In *Proc. ACM 1999 Java Grande Conference*. ACM, June 1999.  
See also [www.wipd.ira.uka.de/~hauma/KaRMI/](http://www.wipd.ira.uka.de/~hauma/KaRMI/)
- [13] OpenMP Architecture Review Board. OpenMP C and C++ Application Programming Interface. Technical report, OpenMP Architecture Review Board, October 1998. Version 1.0. Available from [www.openmp.org](http://www.openmp.org)
- [14] OpenMP Architecture Review Board. OpenMP Fortran Application Programming Interface. Technical report, OpenMP Architecture Review Board, November 1999. Version 1.1. Available from [www.openmp.org](http://www.openmp.org)
- [15] Michael Philippsen and Matthias Zenger. JavaParty — Transparent Remote Objects in Java. In *Concurrency: Practice and Experience*, 9(11):1125-1242, 1997.  
See also [www.wipd.ira.uka.de/JavaParty](http://www.wipd.ira.uka.de/JavaParty)
- [16] Sun Microsystems, Inc. Java Remote Method Invocation Specification. Technical report, Sun Microsystems, Inc., October 1998. Revision 1.50.  
Available at [www.sun.com/research/forest/opj/docs/guide/rmi/spec/rmi-title.doc.html](http://www.sun.com/research/forest/opj/docs/guide/rmi/spec/rmi-title.doc.html)
- [17] Scott Telford. Porting Java RMI 1.2 to an MPI-1 transport. EPCC internal technical note, September 1999.
- [18] K van Reeuwijk, A J C van Gemund, and H J Sips. SPAR: A Programming Language for Semi-automatic Compilation of Parallel Programs. In *Proc. ACM 1997 Workshop on Java for Science and Engineering Computation*. ACM, June 1997.  
See also [www.pds.twi.tudelft.nl/projects/spar](http://www.pds.twi.tudelft.nl/projects/spar)
- [19] Matt Welsh and David Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, December 1999.  
See also [www.cs.berkeley.edu/~mdw/proj/jaguar/](http://www.cs.berkeley.edu/~mdw/proj/jaguar/)
- [20] H W Yau, F E Mourao, M I Parsons, S D Telford, M D Westhead, and N Raj. Parallel Java for the Hitachi SR2201. In *First UK Workshop on Java for High Performance Network Computing at Euro-Par 98*, September 1998. Invited talk.
- [21] K A Yelick, L Semenzato, G Pike, C Miyamoto, B Liblit, A Krishnamurthy, P N Hilfinger, S L Graham, D Gay, P Colella, and A Aiken. Titanium: A High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11-13), 1998.  
See also [www.cs.berkeley.edu/Research/Projects/titanium](http://www.cs.berkeley.edu/Research/Projects/titanium)

## 8 Useful URLs

The following URLs may be of interest, in addition to those listed in the references above:

- Java Grande Forum: [www.javagrande.org](http://www.javagrande.org)
- Java Grande Benchmark Suite: [www.epcc.ed.ac.uk/javagrande](http://www.epcc.ed.ac.uk/javagrande)
- Java for Computational Science and Engineering: [www.npac.syr.edu/projects/tutorials/JavaCSE/](http://www.npac.syr.edu/projects/tutorials/JavaCSE/)
- EuroTools Java Special Interest Group: [www.cscs.ch/Official/SoftwareTech/Java-SIG/](http://www.cscs.ch/Official/SoftwareTech/Java-SIG/)

## A Demonstrator application, sequential Java version

```
public class Inject{

public static int maxloop = 100;
public static double tol = 1.0e-06;

public int m,n,b,h,w;
public double re;
public double psi[ ][ ], zet[ ][ ], u[ ][ ], v[ ][ ];

public Inject(int m, int n, int b, int h, int w, double re)
{
// Constructor for injection grid
  this.m = m;
  this.n = n;
  this.b = b;
  this.h = h;
  this.w = w;
  this.re = re;

  // create arrays (automatically initialised to zero)
  this.psi = new double[n][m];
  this.zet = new double[n][m];
  this.u = new double[n][m];
  this.v = new double[n][m];
}

public void solve()
{
  int loop, redblack, i, j;
  double err, psires, zetres, totres;

  //initial boundary conditions
  boundarypsi();
  boundaryzet();

  loop = 0;
  totres = Double.MAX_VALUE;

  while (loop++ < maxloop && totres > tol) {

    // red black Gauss Seidel iterations

    for (redblack=0;redblack<2;redblack++) {

      for (j=1;j<n-1;j++){
        for (i=1;i<m-1;i++){

          if ((i+j)%2 == redblack) {

            psi[j][i] = 0.25 * (psi[j][i+1]+psi[j][i-1]+
              psi[j+1][i]+psi[j-1][i]-
              zet[j][i]);

            zet[j][i] = 0.25 * (zet[j][i+1]+zet[j][i-1]+
              zet[j+1][i]+zet[j-1][i]) -
              re/16.0 * ((psi[j+1][i]-psi[j-1][i]) *
              (zet[j][i+1]-zet[j][i-1]) -
              (psi[j][i+1]-psi[j][i-1]) *
              (zet[j+1][i]-zet[j-1][i]));

          }
        }
      }

      // impose boundary conditions on zeta

      boundaryzet();

      //compute residuals
```

```

psires = 0.0;
zetres = 0.0;

for (j=1;j<n-1;j++){
    for (i=1;i<m-1;i++){

        err = psi[j][i+1]+psi[j][i-1]+psi[j+1][i]+psi[j-1][i]
            - 4.0 * psi[j][i] - zet[j][i];

        psires += err*err;

        err = zet[j][i+1]+zet[j][i-1]+zet[j+1][i]+zet[j-1][i]
            - 4.0 * zet[j][i]
            - re/4.0*((psi[j+1][i]-psi[j-1][i]) *
                (zet[j][i+1]-zet[j][i-1]) -
                (psi[j][i+1]-psi[j][i-1]) *
                (zet[j+1][i]-zet[j-1][i]));

        zetres += err*err;
    }
}

totres = psires + zetres;

psires = Math.sqrt(psires);
zetres = Math.sqrt(zetres);
totres = Math.sqrt(totres);

if (loop%10 == 0)
    System.out.println("Iteration no. " + loop +
        "    residual = " + totres);
}

System.out.println("Stopped at iteration " + loop +
    "    residual = " + totres);
}

void boundarypsi()
{
    int i,j;

    // bottom
    for (i=0;i<m;i++) psi[n-1][i] = 0.0;

    // top (outflow from aperture)
    for (i=0;i<b;i++) psi[0][i] = 0.0;

    for (i=b;i<b+w-1;i++) psi[0][i] = (double) (i-b+1);
    for (i=b+w-1;i<m;i++) psi[0][i] = (double) w;

    // left
    for (j=0;j<n;j++) psi[j][0] = 0.0;

    //right (inflow from aperture)
    for (j=0;j<h;j++) psi[j][m-1] = (double) w;

    for (j=h;j<h+w-1;j++) psi[j][m-1] = (double) (w-j+h-1);
    for (j=h+w-1;j<n;j++) psi[j][m-1] = 0.0;
}

void boundaryzet()
{
    int i,j;

    // top and bottom
    for (i=0;i<m;i++) {
        zet[0][i] = 2.0 * (psi[1][i] - psi[0][i]);
        zet[n-1][i] = 2.0 * (psi[n-2][i] - psi[n-1][i]);
    }

    //left and right

```

```
    for (j=0;j<n;j++){
        zet[j][0] = 2.0 * (psi[j][1] - psi[j][0]);
        zet[j][m-1] = 2.0 * (psi[j][m-2] - psi[j][m-1]);
    }

public static void main (String[] argv)
{
    Inject inj = new Inject(1000,1000,5,15,5,2.0);

    inj.solve();
}
}
```

## B Demonstrator application, Java threads version

```
public class Inject{

public static int maxloop = 100;
public static double tol = 1.0e-06;

public int m,n,b,h,w;
public double re, residual;
public double psi[[[]], zet[[[]], u[[[]], v[[[]];
public int nthreads;

public Inject(int m, int n, int b, int h, int w, double re, int nthreads)
{
// Constructor for injection grid

    this.m = m;
    this.n = n;
    this.b = b;
    this.h = h;
    this.w = w;
    this.re = re;

    // create arrays
    //initialisation to zero is automatic
    this.psi = new double[n][m];
    this.zet = new double[n][m];
    this.u = new double[n][m];
    this.v = new double[n][m];

    this.nthreads=nthreads;
}

public void solve()
{

    Runnable runners[] = new Runnable[nthreads];
    Thread th[] = new Thread[nthreads];
    Barrier bar = new Barrier(nthreads);
    bar.setMaxBusyIter(1);

    // create threads and grid runner objects

    for(int j=0;j<nthreads;j++){
        runners[j] = new gridrunner(j,nthreads,this,bar);
        th[j] = new Thread(runners[j]);
        th[j].start();
    }

    // wait for threads to finish

    for(int j=0;j<nthreads;j++){
        try{
            th[j].join();
        }
        catch(InterruptedException e){}
    }
}

void boundarypsi()
{
    int i,j;

    // bottom

    for (i=0;i<m;i++) psi[n-1][i] = 0.0;

    // top (outflow from aperture)

    for (i=0;i<b;i++) psi[0][i] = 0.0;

    for (i=b;i<b+w-1;i++) psi[0][i] = (double) (i-b+1);

    for (i=b+w-1;i<m;i++) psi[0][i] = (double) w;
```

```

// left
for (j=0;j<n;j++) psi[j][0] = 0.0;

//right (inflow from aperture)
for (j=0;j<h;j++) psi[j][m-1] = (double) w;

for (j=h;j<h+w-1;j++) psi[j][m-1] = (double) (w-j+h-1);

for (j=h+w-1;j<n;j++) psi[j][m-1] = 0.0;
}

void boundaryzet()
{
    int i,j;

    // top and bottom
    for (i=0;i<m;i++) {
        zet[0][i] = 2.0 * (psi[1][i] - psi[0][i]);
        zet[n-1][i] = 2.0 * (psi[n-2][i] - psi[n-1][i]);
    }

    //left and right
    for (j=0;j<n;j++){
        zet[j][0] = 2.0 * (psi[j][1] - psi[j][0]);
        zet[j][m-1] = 2.0 * (psi[j][m-2] - psi[j][m-1]);
    }
}

public static void main (String[] args)
{
    int nthreads = Integer.parseInt(args[0]);
    Inject inj = new Inject(1000,1000,5,15,5,2.0,nthreads);

    inj.solve();
}
}

class gridrunner implements Runnable{

    int id,nthreads;
    Inject inj;
    Barrier bar;

    public gridrunner(int id, int nthreads, Inject inj, Barrier bar)
    {
        this.id=id;
        this.nthreads=nthreads;
        this.inj = inj;
        this.bar = bar;
    }

    public void run()
    {
        int loop, redblack, i, j, jlo, jhi, slice;
        double err, psires, zetres, totres;
        boolean master;

        master = (id == 0);

        // compute local loop bounds

        slice = (inj.n-2+nthreads-1)/nthreads;
        jlo = id * slice + 1;
        jhi = jlo+slice-1;
        if (jhi > inj.n-2) jhi = inj.n-2;

        //initial boundary conditions

        if (master) {
            inj.boundarypsi();
            inj.boundaryzet();
            inj.residual=Double.MAX_VALUE;

```

```

}
bar.DoBarrier(id);

loop = 0;

while (loop++ < Inject.maxloop && inj.residual > inj.tol) {

    // red black Gauss Seidel iterations

    for (redblack=0;redblack<2;redblack++) {

        for (j=jlo;j<=jhi;j++){
            for (i=1;i<inj.m-1;i++){

                if ((i+j)%2 == redblack) {

                    inj.psi[j][i] = 0.25 * (inj.psi[j][i+1]+inj.psi[j][i-1]+
                        inj.psi[j+1][i]+inj.psi[j-1][i]-
                        inj.zet[j][i]);

                    inj.zet[j][i] = 0.25 * (inj.zet[j][i+1]+inj.zet[j][i-1]+
                        inj.zet[j+1][i]+inj.zet[j-1][i]) -
                        inj.re/16.0 * ((inj.psi[j+1][i]-inj.psi[j-1][i]) *
                        (inj.zet[j][i+1]-inj.zet[j][i-1]) -
                        (inj.psi[j][i+1]-inj.psi[j][i-1]) *
                        (inj.zet[j+1][i]-inj.zet[j-1][i]));

                }
            }
        }
        bar.DoBarrier(id);
    }

    // impose boundary conditions on zeta

    if (master) {
        inj.boundaryzet();
        inj.residual=0.0;
    }
    bar.DoBarrier(id);
    //compute residuals

    psires = 0.0;
    zetres = 0.0;

    for (j=jlo;j<=jhi;j++){
        for (i=1;i<inj.m-1;i++){

            err = inj.psi[j][i+1]+inj.psi[j][i-1]+inj.psi[j+1][i]+inj.psi[j-1][i]
                - 4.0 * inj.psi[j][i] - inj.zet[j][i];

            psires += err*err;

            err = inj.zet[j][i+1]+inj.zet[j][i-1]+inj.zet[j+1][i]+inj.zet[j-1][i]
                - 4.0 * inj.zet[j][i]
                - inj.re/4.0*((inj.psi[j+1][i]-inj.psi[j-1][i]) *
                    (inj.zet[j][i+1]-inj.zet[j][i-1]) -
                    (inj.psi[j][i+1]-inj.psi[j][i-1]) *
                    (inj.zet[j+1][i]-inj.zet[j-1][i]));

            zetres += err*err;
        }
    }

    totres = psires + zetres;

    synchronized(inj){
        inj.residual += totres;
    }

    bar.DoBarrier(id);

    if (master) {
        inj.residual = Math.sqrt(inj.residual);
    }

    if (loop%10 == 0)
        System.out.println(" Iteration no. " + loop +
            "      residual = " + inj.residual);
    }
    bar.DoBarrier(id);

```

```

    }

    if (master){
        System.out.println("Stopped at iteration " + loop +
            "    residual = " + inj.residual);
    }
}

}

public class Barrier {
public volatile int numThreads;

public Barrier(int n) {
    numThreads = n;

    // Initialise the IsDone array. The choice of initial value is
    // arbitrary, but must be consistent!
    IsDone = new boolean[numThreads];
    for(int i = 0; i < numThreads; i++) {
        IsDone[i] = false;
    }
}

public void setMaxBusyIter(int b) {
    maxBusyIter = b;
}

public void DoBarrier(int myid) {
    int b;

    int roundmask = 3;
    boolean donevalue = !IsDone[myid];

    while(((myid & roundmask) == 0) && (roundmask<(numThreads<<2))) {
        int spacing = (roundmask+1) >> 2;
        for(int i=1; i<=3 && myid+i*spacing < numThreads; i++) {
            b = maxBusyIter;
            while(IsDone[myid+i*spacing] != donevalue) {
                b--;
                if(b==0) {
                    Thread.yield();
                    b = maxBusyIter;
                }
            }
        }
        roundmask = (roundmask << 2) + 3;
    }
    IsDone[myid] = donevalue;
    b = maxBusyIter;
    while(IsDone[0] != donevalue) {
        b--;
        if(b==0) {
            Thread.yield();
            b = maxBusyIter;
        }
    }
}

// Array of flags indicating whether the given process and all those
// for which it is responsible have finished. The "sense" of this
// array alternates with each barrier, to prevent having to
// reinitialise.
volatile boolean[] IsDone;
public int maxBusyIter = 1;
}

```

## C Demonstrator application, JOMP version

```
public void solve()
{
    int loop, redblack;
    double err, psires, zetres, totres, lpsires, lzetres;

    //initial boundary conditions

    boundarypsi();
    boundaryzet();

    loop = 0;
    totres = Double.MAX_VALUE;

    while (loop++ < maxloop && totres > tol) {

        // red black Gauss Seidel iterations

        for (redblack=0;redblack<2;redblack++) {

//omp parallel for shared (redblack)
            for (int j=1;j<n-1;j++){
                for (int i=1;i<m-1;i++){

                    if ((i+j)%2 == redblack) {

                        psi[j][i] = 0.25 * (psi[j][i+1]+psi[j][i-1]+
                            psi[j+1][i]+psi[j-1][i]-
                            zet[j][i]);

                        zet[j][i] = 0.25 * (zet[j][i+1]+zet[j][i-1]+
                            zet[j+1][i]+zet[j-1][i]) -
                            re/16.0 * ((psi[j+1][i]-psi[j-1][i]) *
                                (zet[j][i+1]-zet[j][i-1]) -
                                (psi[j][i+1]-psi[j][i-1]) *
                                (zet[j+1][i]-zet[j-1][i]));

                    }
                }
            }

            boundaryzet();

            //compute residuals

            psires = 0.0;
            zetres = 0.0;

//omp parallel for private(err) reduction(+:psires,zetres)
            for (int j=1;j<n-1;j++){
                for (int i=1;i<m-1;i++){

                    err = psi[j][i+1]+psi[j][i-1]+psi[j+1][i]+psi[j-1][i]
                        - 4.0 * psi[j][i] - zet[j][i];

                    psires += err*err;

                    err = zet[j][i+1]+zet[j][i-1]+zet[j+1][i]+zet[j-1][i]
                        - 4.0 * zet[j][i]
                        - re/4.0*((psi[j+1][i]-psi[j-1][i]) *
                            (zet[j][i+1]-zet[j][i-1]) -
                            (psi[j][i+1]-psi[j][i-1]) *
                            (zet[j+1][i]-zet[j-1][i]));

                    zetres += err*err;

                }
            }

            totres = psires + zetres;

            psires = Math.sqrt(psires);
            zetres = Math.sqrt(zetres);
        }
    }
}
```

```
        totres = Math.sqrt(totres);
        if (loop%10 == 0)
            System.out.println("Iteration no. " + loop +
                "    residual = " + totres);
    }
    System.out.println("Stopped at iteration " + loop +
        "    residual = " + totres);
}
```

## D Demonstrator application, mpiJava version

```
import mpi.*;

public class Inject{

public static int maxloop = 100;
public static double tol = 1.0e-06;
public static int nprocs;

public int m,n,b,h,w;
public int myrank,nrows;
public double re;
public double psi[ ][ ], zet[ ][ ], u[ ][ ], v[ ][ ];

public Inject(int m, int n, int b, int h, int w, double re, int myrank)
{
// Constructor for injection grid

    this.m = m;
    this.n = n;
    this.b = b;
    this.h = h;
    this.w = w;
    this.re = re;
    this.myrank = myrank;

// compute number of grid rows per processor

nrows = (n-2)/Inject.nprocs + 2;

// create arrays
this.psi = new double[nrows][m];
this.zet = new double[nrows][m];
this.u = new double[nrows][m];
this.v = new double[nrows][m];
}

public void solve() throws MPIException
{
    int loop, redblack, i, j, globj;
    double err, psires, zetres, totres;

//initial boundary conditions

boundarypsi();
boundaryzet();

loop = 0;
totres = Double.MAX_VALUE;

//halo exchange

haloexchange();

while (loop++ < maxloop && totres > tol) {

// red black Gauss Seidel iterations

for (redblack=0;redblack<2;redblack++) {

for (j=1;j<(nrows-1);j++){

// j is local row number, globj is global row number

globj = myrank * (nrows - 2) + j;

for (i=1;i<m-1;i++){

if ((i+globj)%2 == redblack) {

psi[j][i] = 0.25 * (psi[j][i+1]+psi[j][i-1]+
```

```

        psi[j+1][i]+psi[j-1][i]-
        zet[j][i]);

    zet[j][i] = 0.25 * (zet[j][i+1]+zet[j][i-1]+
        zet[j+1][i]+zet[j-1][i]) -
        re/16.0 * ((psi[j+1][i]-psi[j-1][i]) *
        (zet[j][i+1]-zet[j][i-1]) -
        (psi[j][i+1]-psi[j][i-1]) *
        (zet[j+1][i]-zet[j-1][i]));

    }
}

//halo exchange
haloexchange();
}

boundaryzet();

//compute residuals

psires = 0.0;
zetres = 0.0;

for (j=1;j<(nrows-1);j++){
for (i=1;i<m-1;i++){

    err = psi[j][i+1]+psi[j][i-1]+psi[j+1][i]+psi[j-1][i]
        - 4.0 * psi[j][i] - zet[j][i];

    psires += err*err;

    err = zet[j][i+1]+zet[j][i-1]+zet[j+1][i]+zet[j-1][i]
        - 4.0 * zet[j][i]
        - re/4.0*((psi[j+1][i]-psi[j-1][i]) *
        (zet[j][i+1]-zet[j][i-1]) -
        (psi[j][i+1]-psi[j][i-1]) *
        (zet[j+1][i]-zet[j-1][i]));

    zetres += err*err;
}
}

totres = psires + zetres;

//all reduce totres

double SendBuffer[] = new double [1];
double RecvBuffer[] = new double [1];
SendBuffer[0] = totres;
MPI.COMM_WORLD.Allreduce(SendBuffer,0,RecvBuffer,0,1,MPI.DOUBLE,MPI.SUM);

totres = RecvBuffer[0];

psires = Math.sqrt(psires);
zetres = Math.sqrt(zetres);
totres = Math.sqrt(totres);

if (loop%10 == 0 && myrank == 0)
System.out.println("Iteration no. " + loop +
    "    residual = " + totres);

}

if (myrank == 0)
System.out.println("Stopped at iteration " + loop +
    "    residual = " + totres);

}

void boundarypsi()
{
    int i,j;

    if (myrank == Inject.nprocs - 1){
        for (i=0;i<m;i++) psi[nrows-1][i] = 0.0;
    }
}

```

```

if (myrank == 0) {
    for (i=0;i<b;i++) psi[0][i] = 0.0;
    for (i=b;i<b+w-1;i++) psi[0][i] = (double) (i-b+1);
    for (i=b+w-1;i<m;i++) psi[0][i] = (double) w;
}
for (j=0;j<nrows;j++) psi[j][0] = 0.0;
for (j=0;j<nrows;j++) {
    int globj = myrank * (nrows - 2) + j;
    if (globj < h) {
        psi[j][m-1] = (double) w;
    }
    else{
        if (globj >= h+w-1) {
            psi[j][m-1] = 0.0;
        }
        else {
            psi[j][m-1] = (double) (w-globj+h-1);
        }
    }
}
}

void boundaryzet()
{
    int i,j;
    if (myrank == 0) {
        for (i=0;i<m;i++) zet[0][i] = 2.0 * (psi[1][i] - psi[0][i]);
    }
    if (myrank == Inject.nprocs - 1){
        for (i=0;i<m;i++) zet[nrows-1][i] =
            2.0 * (psi[nrows-2][i] - psi[nrows-1][i]);
    }
    for (j=0;j<nrows;j++){
        zet[j][0] = 2.0 * (psi[j][1] - psi[j][0]);
        zet[j][m-1] = 2.0 * (psi[j][m-2] - psi[j][m-1]);
    }
}

void haloexchange() throws MPIException
{
    // send to neighbour above, receive from below
    // psi first
    if (myrank == 0) {
        MPI.COMM_WORLD.Send(psi[nrows-2],0,m,MPI.DOUBLE,myrank+1,99);
    }
    else {
        if (myrank == Inject.nprocs - 1) {
            MPI.COMM_WORLD.Recv(psi[0],0,m,MPI.DOUBLE,myrank-1,99);
        }
        else {
            MPI.COMM_WORLD.Sendrecv(psi[nrows-2],0,m,MPI.DOUBLE,myrank+1,99,
                psi[0],0,m,MPI.DOUBLE,myrank-1,99);
        }
    }
    // now zet
    if (myrank == 0) {
        MPI.COMM_WORLD.Send(zet[nrows-2],0,m,MPI.DOUBLE,myrank+1,99);
    }
    else {
        if (myrank == Inject.nprocs - 1) {
            MPI.COMM_WORLD.Recv(zet[0],0,m,MPI.DOUBLE,myrank-1,99);
        }
        else {

```

```

        MPI.COMM_WORLD.Sendrecv(zet[nrows-2],0,m,MPI.DOUBLE,myrank+1,99,
                                zet[0],0,m,MPI.DOUBLE,myrank-1,99);
    }
}

// send to neighbour below, receive from above

// psi first

if (myrank == Inject.nprocs - 1) {
    MPI.COMM_WORLD.Send(psi[1],0,m,MPI.DOUBLE,myrank-1,99);
}
else {
    if (myrank == 0) {
        MPI.COMM_WORLD.Recv(psi[nrows-1],0,m,MPI.DOUBLE,myrank+1,99);
    }
    else {
        MPI.COMM_WORLD.Sendrecv(psi[1],0,m,MPI.DOUBLE,myrank-1,99,
                                psi[nrows-1],0,m,MPI.DOUBLE,myrank+1,99);
    }
}

// now zet

if (myrank == Inject.nprocs - 1) {
    MPI.COMM_WORLD.Send(zet[1],0,m,MPI.DOUBLE,myrank-1,99);
}
else {
    if (myrank == 0) {
        MPI.COMM_WORLD.Recv(zet[nrows-1],0,m,MPI.DOUBLE,myrank+1,99);
    }
    else {
        MPI.COMM_WORLD.Sendrecv(zet[1],0,m,MPI.DOUBLE,myrank-1,99,
                                zet[nrows-1],0,m,MPI.DOUBLE,myrank+1,99);
    }
}
}

public static void main (String[] argv) throws MPIException
{
    MPI.Init(argv) ;
    int myrank = MPI.COMM_WORLD.Rank() ;
    nprocs = MPI.COMM_WORLD.Size() ;

    Inject inj = new Inject(1000,1000,5,15,5,2.0,myrank);

    inj.solve();

    MPI.Finalize();
}
}

```

## E Demonstrator application, Fortran OpenMP version

```
program navier2d

implicit none

integer i, j, m, n, loop, maxloop, printfreq, redblack
integer b, h, w

parameter (m=1000, n=1000)
parameter (b=5, h=15, w=5)

parameter (maxloop = 1000)
parameter (printfreq = 10)

double precision psi(m,n), zet(m,n), u(m,n), v(m,n)
double precision re, psires, zetres, totres, err, tol

tol = 1.0d-6

re = 0.2

write(*,*) 'm, n, b, h, w = ', m, n, b, h, w
write(*,*) 're, tol = ', re, tol
write(*,*)

!$omp parallel do default(shared), private(i)

do j = 1, n
do i = 1, m

psi(i,j) = 0.0
zet(i,j) = 0.0

end do
end do

call boundarypsi(psi, m, n, b, h, w)
call boundaryzet(zet, psi, m, n, b, h, w)

do loop = 1, maxloop

do redblack = 0, 1

!$omp parallel do default(shared), private(i)

do j = 2, n-1
do i = 2, m-1

if (mod(i+j,2) .eq. redblack) then

psi(i,j) = 0.25d0*( psi(i+1,j)+psi(i-1,j)
& +psi(i,j+1)+psi(i,j-1)
& -zet(i,j) )
zet(i,j)= 0.25d0*( zet(i+1,j)+zet(i-1,j)
& +zet(i,j+1)+zet(i,j-1) ) -
&
& re/16.0*((psi(i,j+1)-psi(i,j-1)) *
& (zet(i+1,j)-zet(i-1,j)) -
& (psi(i+1,j)-psi(i-1,j)) *
& (zet(i,j+1)-zet(i,j-1)) )

end if

end do
end do

call boundaryzet(zet, psi, m, n, b, h, w)

psires = 0.0
zetres = 0.0

!$omp parallel do default(shared), private(i,err),
!$omp+ reduction(+:psires, zetres)
```

```

do j = 2, n-1
  do i = 2, m-1

    err = psi(i+1,j)+psi(i-1,j)+psi(i,j+1)+psi(i,j-1)
&      -4.0*psi(i,j)-zet(i,j)

    psires = psires + err*err

    err = zet(i+1,j)+zet(i-1,j)+zet(i,j+1)+zet(i,j-1)
&      -4.0*zet(i,j)
&      -re/4.0*((psi(i,j+1)-psi(i,j-1)) *
&              (zet(i+1,j)-zet(i-1,j)) -
&              (psi(i+1,j)-psi(i-1,j)) *
&              (zet(i,j+1)-zet(i,j-1)) )

    zetres = zetres + err*err

  end do
end do

totres = psires + zetres

psires = sqrt(psires)
zetres = sqrt(zetres)
totres = sqrt(totres)

if (totres .lt. tol) then
  write(*,*)
  write(*,*) 'Stopping at loop ', loop, ', residue = ', totres
  goto 999
end if

if (mod(loop,printfreq) .eq. 0) then
  write(*,10) loop, psires, zetres, totres
10  format(1x,'loop ',i5.5,': psires, zetres, res = ',3(g12.4))
  end if

end do

write(*,*)
write(*,20) maxloop
20  format(1x,'+-----+',/,
&      1x,'| No convergence after ', i5.5, ' iterations |', /,
&      1x,'+-----+' )

999  continue

write(*,*) 'Finished'

end

subroutine boundarypsi(psi, m, n, b, h, w)

implicit none

integer m, n, b, h, w, i, j

double precision psi(m,n)

do i = 1, m
  psi(i,n) = 0.0
end do

do i = 1, b
  psi(i, 1) = 0.0
end do

C
C On (b+1:b+w-1,1) we have v=v_y=1, ie dPsi/dx = -1
C

do i = b+1, b+w-1
  psi(i, 1) = real(i-b)
end do

```

```

do i = b+w, m
  psi(i, 1) = real(w)
end do

do j = 1, n
  psi(1,j) = 0.0
end do

do j = 1, h
  psi(m,j) = real(w)
end do

C
C On (m,h+1:w-1) we have u=v_x=1, ie dPsi/dy = 1
C
do j = h+1, h+w-1
  psi(m,j) = real(w-j+h)
end do

do j = h+w, n
  psi(m,j) = 0.0
end do

return
end

subroutine boundaryzet(zet, psi, m, n, b, h, w)

implicit none

integer m, n, b, h, w, i, j

double precision zet(m,n), psi(m,n)

do i = 1, m
  zet(i,1) = 2.0*(psi(i, 2)-psi(i,1))
  zet(i,n) = 2.0*(psi(i,n-1)-psi(i,n))
end do

do j = 1, n
  zet(1,j) = 2.0*(psi(2, j)-psi(1,j))
  zet(m,j) = 2.0*(psi(m-1,j)-psi(m,j))
end do

return
end

```