Towards OpenMP for Java

J. M. Bull and M. D. Westhead
EPCC, University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ,
Scotland, U.K.
email: `[m.bull,m.westhead]@epcc.ed.ac.uk`

M. E. Kambites
Department of Mathematics, University of York,
Heslington, York YO10 5DD, England, U.K.
email: `mek100@york.ac.uk`

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

Abstract

This paper describes JOMP, a definition and implementation of a set of directives and library methods for shared memory parallel programming in Java. A specification of the OpenMP-like directives and methods is proposed. A prototype implementation, consisting of a compiler and a runtime library (both written entirely in Java) is presented, which implements almost all of the proposed specification. Some preliminary performance results are also presented.

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 not too distant future. 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 it what architecture their job will run on. Using Java is not without its problems: perhaps the prime concern for scientific users is performance, though the latest Java compilers are making rapid advances in this field, and are able to run typical scientific kernels at 30-70% of Fortran performance.

It is, of course, possible to write shared memory parallel programs using Java's native threads model [7], [9]. However, a directive system has a number of advantages over the native threads approach. Firstly, the resulting code is much closer to a sequential version of the same program. Indeed, with a little care, it is possible to write an OpenMP program which compiles and runs correctly when the directives are ignored. This makes subsequent development and maintenance of the code significantly easier. It is also to be hoped that, with the increasing familiarity of programmers with OpenMP, parallel programming in Java will become a more attractive proposition.

Another problem with using Java native threads is that for maximum efficiency on shared memory parallel architectures, it is necessary both to use exactly one thread per processor and to keep these threads running during the whole lifetime of the parallel program. To achieve this, it is necessary to have a runtime library which despatches tasks to threads, and provides efficient synchronisation between threads. In particular a fast barrier is crucial to the efficiency of many shared memory parallel programs. Such barriers are not trivial to implement and are not supplied by the `java.lang.Thread` class. Similarly, loop self-scheduling algorithms require careful implementation—in a directive system this functionality is also supplied by the runtime library.

Other approaches to providing parallel extensions to Java include JavaParty [12], HPJava [4], Titanium [15] and SPAR Java [14]. However, these are designed principally for distributed systems, and unlike our proposal, involve genuine language extensions. The current implementations of Titanium and SPAR are via compilation to C, and not Java.

The remainder of this paper is organised as follows: Section 2 discusses the design of the JOMP Application Programmer Interface (API), which is heavily based on the existing OpenMP C/C++ specification. Section 3 describes the JOMP runtime library, a class library which provides the necessary utility routines on top of the `java.lang.Thread` class. Section 4 describes the JOMP compiler, which is written in Java, using the JavaCC compiler building system. In Section 5, we present some preliminary performance results, with comparisons to hand coded Java threads and a commercial Fortran OpenMP implementation. Section 6 concludes, evaluating progress so far.

2 A Draft API

In this section, an informal specification is suggested for an OpenMP-like interface for Java. This is heavily based on the existing OpenMP standard for C/C++ [10], a hence only brief details are presented here.

2.1 Format of Directives

Since the Java language has no standard form for compiler-specific directives, we adopt the approach used by the OpenMP Fortran specification [11] and embed the directives as comments. This has the benefit of allowing the code to function correctly as normal Java: in this sense it is *not* an extension to the language. Another approach would be to use as directives method calls which could be linked to a dummy library. However, this places unpleasant restrictions on the syntactic form of the directives.

A JOMP directive takes the form:

```
//omp <directive> <clauses>
[//omp             <clauses>]
 .....
```

Directives are case sensitive. Some directives stand alone, as statements, while others act upon the immediately following Java code block. A directive should be terminated with a line break. Directives may only appear within a method body. Note that directives may be *orphaned*—work-sharing and synchronisation directives may appear in the dynamic extent of a parallel region of code, note just in its lexical extent.

2.2 The `only` directive

The `only` construct allows conditional compilation. It takes the form:

```
//omp only <statement>
```

The relevant statement will be executed only when the program has been compiled with an JOMP-aware compiler.

2.3 The `parallel` construct

The `parallel` directive takes the form:

```
//omp parallel [if(<cond>)]
//omp [default (shared|none)]
//omp [shared(<vars>)]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
//omp [reduction(<operation>:<vars>)]
 <code block>
```

When a thread encounters such a directive, it creates a new thread team if the boolean expression in the `if` clause evaluates to true. If no `if` clause is present, the thread team is unconditionally created. Each thread in the new team executes the immediately following code block in parallel.

At the end of the parallel block, the master thread waits for all other threads to finish executing the block, before continuing with execution alone.

The `default`, `shared`, `private`, `firstprivate` and `reduction` clauses function in the same way as in the C/C++ standard. The variables may be basic types, or references to arrays or objects, except in the case of the reduction clause, where the variables must be scalars or arrays of basic types.

Note that declaring an object to be `private` causes a new object to be allocated (and initialised with default values) on each thread. Declaring an array to be `private` causes only a new reference to be created on each thread. Declaring an object or array to be `firstprivate` causes a new object or array to be allocated on each thread, which is `cloned` from the existing object or array.

2.4 The `for` and `ordered` directives

The `for` directive specifies that the iterations of a loop may be divided between threads and executed concurrently. The `for` directive takes the form:

```
//omp for [nowait]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
//omp [lastprivate(<vars>)]
//omp [reduction(<operator>:<vars>)]
//omp [schedule(<mode>,[chunk-size])]
//omp [ordered]
 <for loop>
```

As in C/C++, the form of the loop is restricted to so that the iteration count can be determined before the loop is executed. The semantics of this directive and its clauses are equivalent to their C/C++ counterparts. The scheduling `mode` is one of `static`, `dynamic`, `guided` or `runtime`. The `ordered` directive is used to specify that a block of code within the loop body must be executed for each iteration in the order that it would have been during serial execution. It takes the form:

```
//omp ordered
 <code block>
```

2.5 The `sections` and `section` directives

The `sections` directive is used to specify a number of sections of code which may be executed concurrently. A `sections` directive takes the form:

```
//omp sections [nowait]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
//omp [lastprivate(<vars>)]
//omp [reduction(<operator>:<vars>)]
    {
      //omp section
      <code block>
      [//omp section
      <code block>]...
    }
```

The sections are allocated to threads in the order specified, on a first-come-first-served basis. Thus, code in one section may safely wait (but not necessarily busy-wait) for some condition which is caused by a previous section without fear of deadlock.

2.6 The `single` directive

The `single` directive is used to denote a piece of code which must be executed exactly once by some member of a thread team. A `single` directive takes the form:

```
//omp single [nowait]
//omp [private(<vars>)]
//omp [firstprivate(<vars>)]
 <code block>
```

A `single` block within the dynamic extent of a parallel region will be executed only by the first thread of the team to encounter the directive.

2.7 The `master` directive

The `master` directive is used to denote a piece of code which is to be executed only by the master thread (thread number 0) of a team. A `master` directive takes the form:

```
//omp master
 <code block>
```

Unlike the `single` directive, there is no implied barrier at either the beginning or the end of a `master` construct.

2.8 The `critical` directive

The `critical` directive is used to denote a piece of code which must not be executed by different threads at the same time. It takes the form:

```
//omp critical [name]
 <block>
```

Only one thread may execute a critical region with a given name at any one time. Critical regions with no name specified are treated as having the same (null) name. Upon encountering a critical directive, a thread waits until a lock is available on the name, before executing the associated code block. Finally, the lock is released.

2.9 The `barrier` directive

The `barrier` directive causes each thread to wait until all threads in the current team have reached the barrier. It takes the form:

```
//omp barrier
```

To prevent deadlock either all of the threads in a team or none of them must reach the barrier.

2.10 Combined parallel and work-sharing directives

For brevity, two syntactic shorthands are provided for commonly used combinations of directives. The `parallel for` directive defines a parallel region containing only a single `for` construct. Similarly, the `parallel sections` directive defines a parallel region containing only a single `sections` construct.

2.11 Nesting of Directives

The work-sharing directives `for`, `sections` and `single` may not be dynamically nested inside one another. Other nestings are permitted, subject to other stated restrictions concerning what combinations of threads may or may not encounter a construct.

2.12 Library Methods

JOMP provides direct equivalents of all except one of the user-accessible library routines defined in the OpenMP C/C++ standard, implemented as static members of the class `jomp.runtime.OMP`. This includes both simple and nested locks. The exceptional routine is the equivalent of `omp_get_num_procs`, because the number of processors is not available through any standard Java library call. This information could be obtained by making a Java Native Interface call to a system routine, but this would prevent the library from being pure Java. Since the routine is little used, this does not appear to be worthwhile.

`getNumThreads()` returns the number of threads in the team executing the current parallel region, or 1 if called from a serial region of the program.

`setNumThreads(n)` sets to n the number of threads to be used to execute parallel regions. It has effect only when called from within a serial region of the program.

`getMaxThreads()` returns the maximum number of threads which will in future be used to execute a parallel region, assuming no intervening calls to `setNumThreads()`.

`getThreadNum()` returns the number of the calling thread, within its team. The master thread of the team is thread 0. If called from a serial region, it always returns 0.

`inParallel()` returns `true` if called from within the dynamic extent of a parallel region, even if the current team contains only one thread. It returns `false` if called from within a serial region.

`setDynamic()` enables or disables automatic adjustment of the number of threads. `getDynamic()` returns `true` if dynamic adjustment of the number of threads is supported by the OMP implementation and currently enabled. Otherwise, it returns `false`.

`setNested()` enables or disables nested parallelism.

`getNested()` returns `true` if nested parallelism is supported by the OMP implementation and currently enabled. Otherwise, it returns `false`.

2.12.1 The `Lock` and `NestLock` classes

Two types of locks are provided in the library. The class `jomp.runtime.Lock` implements a simple mutual exclusion lock, while the class `jomp.runtime.NestLock` implements a nested lock. Each class implements the same three methods.

The `set()` method attempts to acquire exclusive ownership of the lock. If the lock is held by another thread, then the calling thread blocks until it is released.

The `unset()` method releases ownership of a lock. No check is made that the releasing thread actually owns the lock.

The `test()` method tests if it is possible to acquire the lock immediately, without blocking. If it `is` possible, then the lock is acquired, and the value `true` returned. If it is *not* possible, then the value `false` is returned, with the lock not acquired.

The two lock classes differ in their behaviour if an attempt is made to acquire a lock by the thread which already owns it. In this case, the simple `Lock` class will deadlock, but the `NestLock` class will succeed in reacquiring the lock. Such a lock will be released for acquisition by other threads only when it has been released as many times as it was acquired.

2.13 Environment

Equivalents are provide for all four environment variables defined in the C/C++ standard. They are implemented as Java system properties, which can be set as command line arguments when the Java Virtual Machine is invoked.

The `jomp.schedule` property specifies the scheduling strategy, and optional chunk size, to be used for loops with the `runtime` scheduling option. The form of its value is the same as that used for the parameter to a `schedule` clause.

The `jomp.threads` property specifies the number of threads to use for execution of parallel regions.

The `jomp.dynamic` property takes the value `true` or `false` to enable or disable respectively dynamic adjustment of the number of threads.

The `jomp.nested` property takes the value `true` or `false` to enable or disable respectively nested parallelism.

2.14 Differences from C/C++ standard

The main differences from the C/C++ standard are as follows:

- The `threadprivate` directive, and hence the `copyin` clause, are not supported. Java has no global variables, as such. The only data to which such a concept might be applied are static class members, but this is both unattractive and difficult to implement.

- The `atomic` directive is not supported. The kind of optimisations which the directive is designed to facilitate (for example, atomic updates of array elements) require access to atomic test-and-set instructions which are not readily available in Java. The `atomic` directive would merely be a synonym for the `critical` directive.

- The `flush` directive is not supported, since it also requires access to special instructions. Provided that variables used for synchronisation are declared as `volatile`, this should not be a problem. However, it is not clear how the ambiguities in the Java memory model specification noted in [13] affect this issue.

- Array reductions are permitted.

- There is no function to return the number of processors.

3 The JOMP runtime library

As well as the user-accessible methods, the package `jomp.runtime` contains a library of classes and routines used by compiler-generated code. We describe them only briefly here—for more details see [2] and [6].

The core of the library is the `OMP` class. As well as the user-accessible methods, this class contains the routines used by the compiler to implement parallelism in terms of Java's native threads model.

The `BusyThread` and `BusyTask` classes are used for thread-management purposes. Tasks to be executed in parallel are instances of the class `BusyTask`. They have a single method, `go()`, which takes as a parameter the number (within its team) of the executing thread. All threads except the master are instances of the class `BusyThread`, which extends `Thread` and has a `BusyTask` reference as a member. Each non-master thread executes a loop, in which it reaches a global barrier, executes its task, and then reaches the barrier again. The loop may be terminated (after the first barrier call) on the setting of a flag by the master thread.

The barrier is provided by the `Barrier` class which implements a static 4-way tournament barrier [5] for an arbitrary number of threads. This is a lock-free algorithm whose correctness cannot be formally guaranteed under the current specification of the Java memory model. However, we have observed no such problems in practice. This class is also used for the `barrier` directive and implied barriers in other directives. Critical sections are implemented as `synchronized` blocks. The `OMP` class stores a hash table of lock objects in order to implement named critical sections.

The `Reducer` class is used to implement the `reduction` clause. It provides methods for the different reduction operators on different types, and uses the same tournament algorithm as the `Barrier` class.

Work sharing is facilitated by the `Ticketer` class. For `sections` and `ordered` directives a `Ticketer` object issues integer tickets, in sequence. To support the `for` directive, a `Ticketer` object issues loop chunks according to the different loop scheduling schemes. The `Orderer` class is used to implement the `ordered` construct. It stores, and controls access to, the next iteration of a loop to be executed.

Nested parallelism is not currently supported, as is generally the case in current implementations of the OpenMP C/C++ and Fortran specifications. If the `doParallel()` method is called by a thread in parallel mode, thread-specific data is copied, the thread is reconfigured to be in its own team of size one, and the task is executed. Finally, the original values of the thread-specific data are restored.The `setNested()` method does nothing, and the `getNested()` method always returns `false`.

4 The JOMP Compiler

The JOMP Compiler is built around a Java 1.1 parser provided as an example with the JavaCC [8] utility. JavaCC comes supplied with a grammar to parse a Java 1.1 program into a tree, and an `UnparseVisitor` class, which unparses the tree to produce code. The bulk of the compiler is implemented in the `OMPVisitor` class, which extends the `UnparseVisitor` class, overriding various methods which unparse particular nonterminals. Because JavaCC is itself written in Java, and outputs Java source, the JOMP system is fully portable, and requires only a JVM installation in order to run it. These overriding methods output modified code, which includes calls to the runtime library to implement appropriate parallelism.

Upon encountering a `parallel` directive within a method, the compiler creates a new class. If the `default(shared)` clause is specified, an inner class (within the class containing the current method) is created. If the method containing the `parallel` directive is `static` then the new inner class is also `static`. If `default(none)` is used, then a separate class within the same compilation unit is created. For each variable declared to be `shared`, the class contains a field of the same type signature and name. For each variable declared to be `firstprivate` or `reduction`, the class contains a field of the same type signature and a local name.

The new class has a single method, `go`, which takes a parameter indicating an absolute thread identifier. For each variable declared to be `private`, `firstprivate` or `reduction`, the `go()` method declares a local variable with the same name and type signature. The local `firstprivate` variables are initialised from the corresponding field in the containing class, while the local `private` variables have default initialisation. The local `reduction` variables are initialised with the appropriate default value for the reduction operator. Private objects are allocated using the default constructor. The main body of the `go()` method contains the code to be executed in parallel.

In place of the parallel construct itself, code is inserted to declare a new instance of the compiler created class, and to initialise the fields within it from the appropriate variables. The `OMP.doParallel()` method is used to execute the `go` method of the inner class in parallel. Finally, any values necessary are copied from class fields, back into local variables. Figures 1 and 2 illustrate this process for a trivial "Hello World" program.

Work sharing and synchronisation directives are implemented by adding code which utilises calls to the runtime library described in Section 3. For further details see [2].

```
import jomp.runtime.*;
public class Hello {
  public static void main (String argv[]) {
    int myid;
//omp parallel private(myid)
    {
      myid = OMP.getThreadNum();
      System.out.println("Hello from " + myid);
    }
  }
}
```

Figure 1: “Hello World” JOMP program

```
import jomp.runtime.*;
public class Hello {
  public static void main (String argv[]) {
    int myid;
    __omp_class_0 __omp_obj_0 = new __omp_class_0();
    try {
      jomp.runtime.OMP.doParallel(__omp_obj_0);
    }
    catch(Throwable __omp_exception) {
      System.err.println("OMP Warning: exception
                          in parallel region");
    }
  }
  private static class __omp_class_0
        extends jomp.runtime.BusyTask {
    public void go(int __omp_me) throws Throwable {
      int myid;
      myid = OMP.getThreadNum();
      System.out.println("Hello from " + myid);
    }
  }
}
```

Figure 2: Resulting “Hello World” Java program

5 Performance analysis

To examine the performance of the JOMP system, a simple simulation of two dimensional fluid flow in a box was used. The 2-D steady-state viscous 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. As well as a JOMP version, a hand-coded Java threads version and version using mpiJava [1] (a Java interface to a native MPI library) have been written. The JOMP version differs from the sequential version only by the addition of two `parallel for` directives.

The three parallel Java versions of this code were run on a Sun HPC 6500 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 was also tested. (This was compiled with the KAI guidef90 compiler, Version 3.7 and then the Sun WorkShop 5.0 f90 compiler, with flags `-fast -xarch=v8plusa`.) In all cases, 100 red-black iterations were executed on a 1000×1000 grid.

Figure 3 shows the performance of the versions of the codes compared to ideal speedup calculated from the performance of sequential Fortran and Java versions.

The results show that the hand coded Java threads version gives the best performance of the three Java versions, showing some slight superlinear speedup. The JOMP version gives on slightly lower performance, and also scales well. The mpiJava version shows some stronger superlinearity up to eight processors, but scales poorly on larger numbers of processors. The Java versions attain approximately half the performance of the Fortran OpenMP version.

We have also compared the overheads of the synchronisation constructs in the JOMP runtime library to those of guidef90. The methodology consists of comparing the time taken to execute the same code with and without each directive, and is described fully in [3]. However, these results should be interpreted with care, as microbenchmarks can exhibit odd behaviour with just-in-time compliers. Table 1 shows the overhead of the various constructs on 16 processors on the Sun HPC 6500.

Within the exception of the `single` directive, all the directives JOMP directives requiring barrier synchronisation outperform those their Fortran equivalents under guidef90, as the basic barrier routine is approximately four times faster. The reason for the very high overhead of the `single` directive is not clear: in the microbenchmark, synchronised accesses to a `Ticketer` object are made immediately following a barrier, which may result in heavy contention.

The overheads of the locking type synchronisation are somewhat higher in JOMP than in guidef90. In JOMP, the overhead of these directives are determined by the cost of Java `synchronized` blocks, and there is little scope for optimising the performance of these directives any further.

Directive	guidef90	JOMP
PARALLEL	78.4	34.3
PARALLEL + REDUCTION	166.5	58.4
DO/FOR	42.3	24.6
PARALLEL DO/FOR	87.2	42.9
BARRIER	41.7	11.0
SINGLE	83.0	1293
CRITICAL	11.2	19.1
LOCK/UNLOCK	12.0	20.9
ORDERED	12.4	47.0

Table 1: Synchronisation overheads (in microseconds) on Sun HPC 6500

6 Conclusions and Future Work

We have defined an OpenMP-like interface for Java which enables a high level approach to shared memory parallel programming. A prototype compiler and runtime library which implement most of the interface have been described, showing that the approach is feasible. Only minor changes from the OpenMP C/C++ specification are required, and the implementation of both the runtime library and the compiler are shown to be relatively straightforward. Initial analysis shows that the resulting code scales well, with little overhead compared to a hand-coded Java threads version. Low-level synchronisation overheads have been measured and are for the most part, tolerable.

Further work includes producing a complete specification, taking particular care with scoping issues and restrictions. While portability of functionality should not be an issue, portability of performance is of more concern, and should be examined across different platforms and different virtual machines. A small amount of the specification (predominantly support for nested parallelism, and array reductions) remains to be implemented.

References

[1] M. Baker, B. Carpenter, G. Fox, S.-H. Ko, and S 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.

[2] J. M. Bull and M. E. Kambites. JOMP – an OpenMP-like Interface for Java. In *Proceedings*

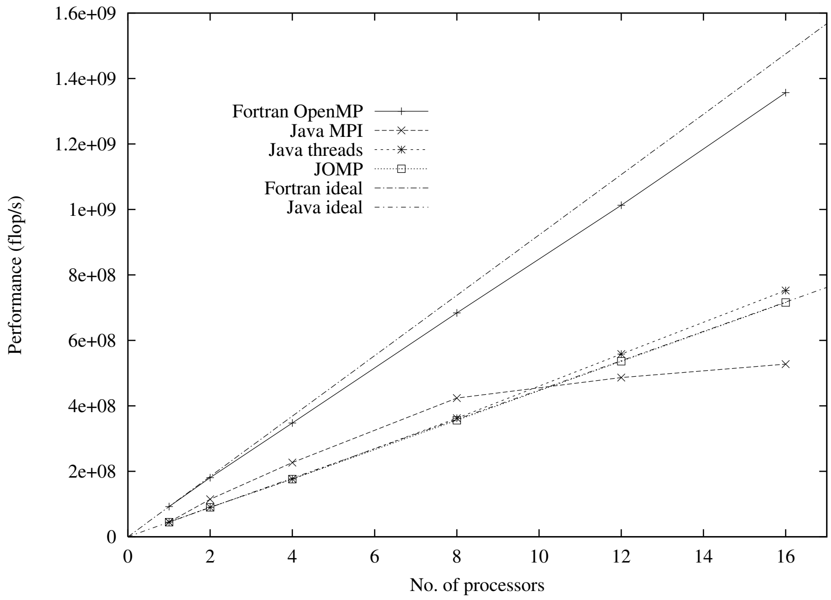

Figure 3: Performance of CFD application on Sun HPC 6500

of the ACM 2000 Java Grande Conference, June 2000, pp. 44–53.

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

[4] B. Carpenter, G. Zhang, G. Fox, X. Li and Y. Wen. HPJava: Data Parallel Extensions to Java. *Concurrency: Practice and Experience*, 10(11-13):873-877, 1998.

[5] D. Grunwald and S. Vajracharya. Efficient Barriers for Distributed Shared Memory Computers. In *Proceedings of 8th International Parallel Processing Symposium*, April 1994.

[6] M. E. Kambites. Java OpenMP: Demonstration implementation of a compiler for a subset of OpenMP for Java, EPCC Summer Scholarship Programme Technical Report, September 1999, available from `www.epcc.ed.ac.uk/ssp/1999/ProjectSummary/kambites.html`.

[7] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.

[8] Metamata Inc. JavaCC—The Java Parser Generator. `www.metamata.com/JavaCC`.

[9] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 1997.

[10] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 1.0. Available from `www.openmp.org`, October 1998.

[11] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.1. Available from `www.openmp.org`, November 1999.

[12] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[13] W. Pugh. Fixing the Java Memory Model. In *Proceedings of ACM 1999 Java Grande Conference*, pages 89–98. ACM Press, June 1999.

[14] K. van Reeuwijk, A. J. C. van Gemund, and H. J. Sips. SPAR: A Programming Language for Semi-automatic Compilat ion of Parallel Programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, 1997.

[15] 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):825–836, 1998.