An OpenMP-like Interface for Parallel Programming in Java

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

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

J. M. Bull
Edinburgh Parallel Computing
Centre,
University of Edinburgh,
Mayfield Road, Edinburgh
EH9 3JZ, Scotland, U.K.
m.bull@epcc.ed.ac.uk

ABSTRACT

This paper describes the definition and implementation of an OpenMP-like set of directives and library routines for shared memory parallel programming in Java. A specification of the directives and routines is proposed and discussed. A prototype implementation, consisting of a compiler and a runtime library, both written entirely in Java, is presented, which implements most of the proposed specification. Some preliminary performance results are reported.

1. INTRODUCTION

OpenMP is a relatively new industry standard for shared memory parallel programming, which is enjoying increasing levels of support from both users and vendors in the high performance computing field. The standard defines a set of directives and library routines for both Fortran [13] and C/C++ [12], and provides a higher level of abstraction to the programmer than, for example, programming with POSIX threads [6]. On the other hand, OpenMP programs can be less scalable and less portable than message-passing programs.

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, a directive system will make parallel programming in Java 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, but may be two or three orders of magnitude cheaper than thread fork/joins. Similarly, loop self-scheduling algorithms require careful implementation—in a directive system this functionality is also supplied by the runtime library. These concerns could be met without recourse to directives, simply by supplying the appropriate class library. Another possible approach, therefore, would be to modify the run-time library described here for direct use by the programmer.

Other approaches to providing parallel extensions to Java include JavaParty [15], HPJava [3], Titanium [19] and SPAR Java [18]. 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 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 translates Java with JOMP directives to pure Java with calls to the JOMP runtime library. In Section 5, we present some preliminary performance results, with comparisons to hand coded Java threads and a commercial Fortran OpenMP implementation. Section 6 raises some outstanding issues, which would benefit from further research, while Section 7 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 draft is heavily based on the existing OpenMP standard for C/C++ [12], and hence only brief details are presented here. A more comprehensive specification is given in [11].

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 [13] 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, not 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 `clone`d from the existing object or array.

Example: Computing the sum of a two-dimensional array where each thread has one row.

```
//omp parallel shared(a,n) private(myid,i)
//omp          reduction(+:b)
 {
    myid = OMP.getThreadNum();
    for (i=0; i<n; i++) {
        b += a[myid][i];
    }
 }
```

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 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>
```

Example: Simple parallel loop.

```
//omp parallel shared(a,b)
  {
//omp for
     for (int i=1; i<n; i++){
          b[i] = (a[i] + a[i-1]) * 0.5;
     }
  }
```

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.

Example: Independent methods.

```
//omp parallel shared(a,b,c)
  {
//omp sections
     {
//omp section
       a.init();
//omp section
       b.init();
//omp section
       c.init();
     }
  }
```

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. Other threads skip the block and wait at a barrier for the execution of the block to complete. 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.

Example: First thread to finish initialisation reads some data in.

```
//omp parallel
  {
    x.init();
//omp single
    {
      y.readin();
    }
    doWork(x,y);
  }
```

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.

Example: Simple I/O.

```
//omp parallel
  {
    doWork();
//omp master
    { System.out.println(" some output here "); }
    doMoreWork();
  }
```

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 JOMP implementation and currently enabled. Otherwise, it returns `false`.

`setNested()` enables or disables nested parallelism.

`getNested()` returns `true` if nested parallelism is supported by the JOMP 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 available 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 [16] affect this issue.

- Array reductions are permitted. This feature has been added to OpenMP Fortran standard in Version 2.0 [14] and seems likely to be added to the C/C++ standard in the future.
- There is no function to return the number of processors, as noted in Section 2.12.

3. THE JOMP RUNTIME LIBRARY

In this section we describe the JOMP runtime library, which provides the necessary functionality to support parallelism in terms of Java's native threads model. Further details of the implementation are given in in [5] and [10].

3.1 Structure of the Library

As well as the user-accessible functions and locks specified in Section 2.12, the package `jomp.runtime` contains a library of classes and routines used by compiler-generated code.

The core of the library is the `OMP` class. As well as the user-accessible functions documented in Section 2.12, 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. The `Barrier` class implements a barrier, and is used for internal thread-management purposes, as well as for implementing the directives which require this construct. The `Orderer` class is used to facilitate implementation of the `ordered` construct, while the `Reducer` class implements reductions of variables. The `Ticketer` and `LoopData` classes are used to facilitate scheduling. The `Lock` and `NestLock` classes implement the user-accessible locks described in Section 2.12.1.

3.2 A Question of Personal Identity

In order that threads can perform different tasks, it is necessary that the code they execute has some way of distinguishing between them. The need to support orphaned directives (see Section 2) means that it is not sufficient simply to give each thread a private variable indicating its identity. Upon encountering an orphaned directive, the variable may no longer be in scope. The only variables which will certainly be in scope are static class fields. Unfortunately, the values taken by these are by nature common to all threads, and so cannot be used to differentiate between them.

Nor can we simply pass an ID down the dynamic call chain, as an extra parameter for each function. Apart from the complexity involved in deciding which functions need such parameters and which do not, there is no guarantee that the call chain does not encompass functions for which the source code is not available.

The only way to distinguish between threads is by use of the static `currentThread()` method of the `Thread` class, which returns a reference to the appropriate instance of the `Thread` class. It would be nice to give our `BusyThread` class an integer field in which to store its own ID. Unfortunately, the master thread is not an instance of `BusyThread`. One approach would be to perform a runtime type check on the `currentThread()`, and assume that we are in the master thread if we cannot cast to type `BusyThread`.

We can circumvent this problem by storing an absolute numerical ID for each process, in ASCII decimal format, in the process name field. The library `getAbsoluteID()` call simply parses the name field of the `currentThread()`. This is evidently not very efficient, but we can reduce performance impact by minimising the number of calls to `getAbsoluteID()`.

To facilitate this, many of the methods in the library have two versions, one of which takes as an extra (first) parameter the absolute process ID of the calling thread.

3.3 Initialisation

Initialisation is divided into two parts. The static initialisation for the class `jomp.runtime.OMP` reads the system properties documented in Section 2.13. These are used to set up the numbers of threads to use, and to set up the static subclass `Options`, which contains configuration information.

The `start()` method is called on demand, when the first parallel region is encountered. It initialises the critical region table (see Section 3.11) and all the thread-specific data. It then creates a team of threads and sets them running, whereupon they wait to be assigned a task.

3.4 Tasks and Threads

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 but 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 reaches the barrier again. The loop may be terminated (after the first barrier call) on the setting of a flag by the master thread.

During execution of serial regions of the program, the threads all pause at the first barrier in the loop, waiting for the master thread to reach the barrier. When the master thread calls the `doParallel()` method, it sets up the tasks of each thread and reaches the global barrier, thus causing the other threads to execute the task. The master then executes the task in its own right, before reaching the barrier again, causing it to wait for all other threads to finish parallel execution before continuing with serial execution alone.

All but the master thread are *daemon* threads, so that they die if the master thread terminates. The implicit barrier at the end of every parallel region ensures that the master thread cannot terminate while the others are doing useful work.

The thread scheduling policy is largely the responsibility of the operating system. In almost all circumstances, the number of threads used to execute a parallel program should not exceed the number of available processors. In order to prevent the possibility threads from tying up resources indefinitely, threads waiting at a barrier will eventually yield—see Section 3.6.

3.5 Nested Parallelism

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`.

3.6 Barriers

The `Barrier` class implements a simple, static 4-way tournament barrier [4] for an arbitrary number of threads. Its constructor takes as a parameter the number of threads to use.

The `doBarrier()` method takes as a parameter a thread number, and causes the calling thread to block until it has been called the same number of times for each possible thread number.

To avoid the overhead of a system call, threads busy-wait. Unfortunately, many Java systems implement co-operative rather than pre-emptive multitasking. If the threads are not each allocated their own processor, busy-waiting can cause deadlock. To avoid this, a thread busy-waits by going around an empty loop a set number of times, before `Yielding` to other threads. The number of iterations performed before yielding can be set by calling the `setMaxBusyIter()` method, and can be tuned for different systems.

The `OMP` class maintains, for each thread, a `Barrier` reference pointing to a single barrier for each team. The `OMP.doBarrier()` method reaches the appropriate barrier for the calling thread.

3.7 Reductions

The `Reduction` class is used to implement the `reduction` clause. A call to a reduction method causes the calling thread to wait until all other threads have called the routine with their respective values. The method then returns the result of the reduction. The `Reducer` is implemented using a static 4-way tournament algorithm, in almost exactly the same way as the `Barrier`.

The `OMP` class maintains a `Reducer` reference for each thread, which points to a common `Reducer` for the team. Calls to the different `OMP.do...Reduce()` methods from within a parallel region are passed to the relevant method in the appropriate `Reducer`. During serial execution, the calls simply return their argument.

3.8 Scheduling

3.8.1 *The* `LoopData` *class*

A `LoopData` object is used to store information about a loop or a chunk of a loop. It contains details of the start, step and stop of a loop. The stop value is stored so as to make the loop continuation expression a strict inequality. The object also contains a field to indicate the chunk size to be used when dividing up the loop. In addition, it contains a secondary step value, which allows a `LoopData` object to represent a set of chunks, evenly spaced throughout a loop. Finally, there is a flag to indicate whether a chunk is the last which could be executed by the calling thread.

3.8.2 *The* `Ticketer` *class*

The `Ticketer` class is used to facilitate dynamic allocation of work to different threads.

The synchronized `issue()` method is used to issue tickets: successive calls return integer tickets, starting at zero. This facility is used to implement the `single` and `sections` constructs. Calls to the `issueDynamic()` and `issueGuided()` methods issue successive chunks of a loop, using simple first-come-first-served, and guided self-scheduling, strategies respectively.

The `reset()` method returns the next in a conceptually infinite list of ticketers, to be used for the next operation. This allows a thread with no work to begin executing the next work-sharing construct without waiting for its peers.

3.9 Ordering Support

The `Orderer` class is used to implement the `ordered` construct. It stores, as its state, the next iteration of a loop to be executed. The `reset()` method takes a loop counter value indicating the first iteration of the following loop, and returns the next in a conceptually infinite list of `Orderer`s.

The `startOrdered()` method blocks until the given loop iteration is the next to be executed, and then returns. The `stopOrdered()` method sets the next iteration indicator to the given value. The `OMP` class maintains for each thread a reference into a conceptually infinite list of `Orderer`s. The `startOrdered()` and `stopOrdered()` methods pass their parameters on to the appropriate methods of the relevant `Orderer`.

The `resetOrderer()` method advances the thread's reference to point to the next `Orderer`, setting up the value of the first iteration if it is not already set. When all threads have advanced past an `Orderer`, no reference to the object remains, and so it will be available for garbage collection.

3.10 Locks

The `Lock` and `NestLock` classes described in Section 2.12.1 are implemented in a straightforward manner, using the Java `synchronized` method modifier to provide mutual exclusion.

3.11 Critical Regions

The requirement that names of critical regions be global in scope presents a problem. JOMP directives are to be replaced by Java code, so we need some construct in Java which allows us to access the same lock regardless of the current scope.

One approach would be to create a public *class* for each critical region name, in a predetermined place in the class hierarchy—say `jomp.runtime.critical`. Such a class would have static members to facilitate locking. However, the requirement imposed by Java compilers that such classes occupy a predetermined place in the directory structure may cause problems. Quite apart from the obvious messiness, there is no guarantee that the user will have permission to write to the appropriate location!

Instead, we choose a neater, if less efficient, solution. The `OMP` class maintains, as a static member, a hash table, indexed by name and containing, for each name, an instance of class `Lock`. The structured block associated with the directive is enclosed in a *synchronized* statement. Locks passed as a parameter are held in a static hash table and the `getLockByName` method is used to get a reference to the lock associated with a given name, creating it if necessary.

4. THE JOMP COMPILER

In this section, we describe a simple compiler which implements a large subset of the specification suggested above. Additional implementation details may be found in [5] and [10]. Currently, a few parts of the specification have yet to be implemented, such as nested parallelism and array reductions.

4.1 Basic Structure

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.

4.2 Personal Identity Revisited

As discussed in Section 3.2, there is no cheap way for a thread to identify itself. To alleviate this problem, the compiler creates code which attempts to keep track of its own ID, in the variable `__omp_me`.

Where `__omp_me` is not in scope, and library calls are inserted which might entail in multiple calls to `getAbsoluteID()`, code is inserted to declare `__omp_me` and initialise it to the value returned by a call to `getAbsoluteID()`. The `isMeDefined` flag is set in the compiler, to provide information for visitors within the static scope of the new declaration. Where a library call would entail a single call to `getAbsoluteID()`, the value of `__omp_me` is used if available.

For simplicity, these technicalities are largely ignored in the sections that follow, and all library calls are shown without their thread number parameters.

In the current JOMP implementation, the compiler neither performs a full semantic analysis, nor keeps a track of package, classes, variables and its names, with a single exception of local variables. It does not even keep a track of the current class's fields. It simply works with one compilation unit at a time, and relies on the programmer to provide type information in data attribute clauses.

4.3 The `parallel` directive

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.

A very simple "Hello World" example illustrating this process is shown in Figures 1 and 2.

4.4 Work-sharing directives

Upon encountering the `for`, `sections`, or `single` directive, a new block is created. For each variable declared to be `firstprivate`, a local variable `_fp_<varname>` is declared and initialised by the value of the original variable. For each variable declared to be `lastprivate`, a local variable `_lp_<varname>` is declared. For each variable declared to be `reduction`, a local variable `_rd_<varname>` is declared. These newly created variables are used to communicate the values of variables to the enclosing block. In the case of the `for` and `sections` directives, the `__omp_amLast` boolean variable is declared to hold information about whether the current thread is the one performing the sequentially last iteration of the loop, or the sequentially last `section`.

Inside the newly allocated block, a second block is created. For each variable declared to be `firstprivate, private, lastprivate,` or `reduction`, a new variable with the same

```
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

```
public class Hello {
  public static void main (String argv[]) {
    int myid;
    // OMP PARALLEL BLOCK BEGINS
    {
      __omp_Class0 __omp_Object0 = new __omp_Class0();
      __omp_Object0.argv = argv;
      try {
        jomp.runtime.OMP.doParallel(__omp_Object0);
      } catch(Throwable __omp_exception) {
        System.err.println("OMP Warning: Illegal thread exception ignored!");
        System.err.println(__omp_exception);
      }
      argv = __omp_Object0.argv;
    }
    // OMP PARALLEL BLOCK ENDS
    }
    // OMP PARALLEL REGION INNER CLASS DEFINITION BEGINS
    private static class __omp_Class0 extends jomp.runtime.BusyTask {
      String [ ] argv;
      public void go(int __omp_me) throws Throwable {
      int myid;
        // OMP USER CODE BEGINS
        {
          myid = OMP.getThreadNum();
          System.out.println("Hello from " + myid);
        }
        // OMP USER CODE ENDS
      }
    }
    // OMP PARALLEL REGION INNER CLASS DEFINITION ENDS
}
```

Figure 2: Resulting "Hello World" Java program

name is declared. Variables declared to be `reduction` are initialised with the appropriate value. `private` and `lastprivate` variables are initialised by calling the default constructor in the case of class type variables, and uninitialised in the case of primitive or array type variables. `firstprivate` variables are initialised by the appropriate value from the `_fp_` copy of the original variable. A `clone()` method is called to initialise class or array type variables.

Next, the code to actually handle the appropriate worksharing directive is inserted (see below). At the end of the inner block appropriate local variable (`_lp_<varname>` or `_rd_<varname>`) is updated for every `lastprivate` and `reduction` variable.

After the end of the inner block, a code to update the global copies of `lastprivate` and `reduction` variables is inserted. `lastprivate` variables are updated only by the thread with the variable `__omp_amLast` set to TRUE. Reduction variables are updated by the master thread of the team. Finally, the outer block is closed.

Figures 3 and 4 illustrate this process for a simple parallel loop.

4.4.1 *The* `for` *directive*

Upon encountering a `for` directive, the compiler inserts code to create two `LoopData` structures. One of these is initialised to contain the details of the whole loop, while the other is used to hold details of particular chunks. The generated code then repeatedly calls the appropriate `getLoop...()` function for the selected schedule, executing the blocks it is given, until there are no more blocks. If a dynamic scheduling strategy was used, the ticketer is then reset. Any reductions are carried out, and if the `nowait` clause is not specified, the `doBarrier()` method is called.

4.4.2 *The* `ordered` *clause and directive*

If the `ordered` clause is specified on a `for` directive, then a call to `resetOrderer()` is inserted immediately prior to the loop, at which point the value of the first iteration number is definitely known.

Upon encountering an `ordered` directive, the compiler inserts a call to `startOrdered()` before the relevant block with the parameter being the current value of the loop counter. After the block is inserted a call to `stopOrdered()`, with the parameter being the next value the loop counter would take *after* its current value, during sequential execution:

```
jomp.runtime.OMP.startOrdered(i);
<block>
jomp.runtime.OMP.stopOrdered(i+step);
```

4.4.3 *The* `master` *directive*

Upon encountering a `master` directive, the compiler inserts code to execute the relevant block if and only if the `OMP.getThreadNum()` method returns 0.

```
if(jomp.runtime.OMP.getThreadNum()==0) {
    <block>
}
```

4.4.4 *The* `single` *directive*

Upon encountering a `single` directive, the compiler inserts code to get a ticket, execute the relevant block if and only if the ticket is zero, and then reset the ticketer. If the `nowait` clause is not specified, the `doBarrier()` method is called.

```
if(jomp.runtime.OMP.getTicket()==0) {
    <block>
}
jomp.runtime.OMP.resetTicket();
[jomp.runtime.OMP.doBarrier();]
```

4.4.5 *The* `sections` *directive*

Upon encountering a `sections` directive, the compiler inserts code which repeatedly requests a ticket from the ticketer, and executes a different section depending on the ticket number. When there are no sections left, the ticketer is reset. If the `nowait` clause is not specified, the `doBarrier()` method is called.

```
some_label : for(;;) {
    switch(jomp.runtime.OMP.getTicket()) {
        case 0 : <section 0>; break;
        case 1 : <section 1>; break;
        case 2 : <section 2>; break;
        default : break some_label;
    }
}
jomp.runtime.OMP.resetTicket();
[jomp.runtime.OMP.doBarrier();]
```

4.5 Synchronisation directives

4.5.1 *The* `critical` *directive*

Upon encountering a `critical` directive, the compiler creates a synchronized block, with a call to `getLockByName`.

```
synchronized
(jomp.runtime.OMP.getLockByName("name"))
{
 <block>
}
```

4.5.2 *The* `barrier` *directive.*

Upon encountering a `barrier` directive, the compiler inserts a call to the `doBarrier()` method.

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.

```
//omp for firstprivate(i) private(j) lastprivate (k) reduction(+:l)
for(int m=0; m<100;m++)
  ...
```

Figure 3: Fragment of JOMP program

```
{ // OMP FOR BLOCK BEGINS
  // copy of firstprivate variables, initialised
  int _fp_i = i;
  // copy of lastprivate variables
  int _lp_k;
  // variables to hold result of reduction
  int _rp_l;
  boolean __omp_amLast=false;
  {
    // firstprivate variables + init
    int  i = (int) _fp_i;
    // [last]private variables
    int j;
    int k;
    // reduction variables + init to default
    int  l = 0;

     ... code to execute the parallel loop, perform
         reduction into _rp_l and set __omp_amLast ...

    // copy lastprivate variables out
    if (__omp_amLast) {
      _lp_k = k;
    }
  }
  // set global from lastprivate variables
  if (__omp_amLast) {
    k = _lp_k;
  }
  // set global from reduction variables
  if (jomp.runtime.OMP.getThreadNum(__omp_me) == 0) {
    l+= _rp_l;
  }
} // OMP FOR BLOCK ENDS
```

Figure 4: Resulting Java code

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 5 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 only 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 [2]. 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.

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

With 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.

Further performance studies, which show that JOMP code generally gives performance comparable to hand coded Java threads, can be found in [10].

6. OUTSTANDING ISSUES

In this section, we briefly outline some of the outstanding issues which have yet to be resolved, and which require more work.

6.1 The `atomic` directive

As noted in Section 2.14, this directive cannot readily be implemented if it has comparable semantics to the C/C++ equivalent. Alternative semantics for this directive, for example atomic updates at the object level, are worth considering. They would be more object-oriented in nature and easier to implement in practice.

6.2 Semantic analysis

A compiler performing full semantic analysis (or at least full analysis of *names*) is required. Having information about all the package/class/method names, and about binding identifiers to particular variables, would made the JOMP compiler more straightforward to implement. The treatment of data scope attribute clauses, in particular, would be simplified.

6.3 Error handling

The JOMP compiler has minimal error checking capability. In practice, it is necessary to ensure that a program compiles correctly with the sequential compiler before attempting to run the JOMP compiler on it. Some error checking for the directive syntax should be added.

6.4 Exception Handling

Exceptions are an important feature of the Java language, and it is worth considering how they will be handled by an OpenMP-like implementation. Exceptions are present in C++, but they are less widely used than in Java and the OpenMP C/C++ specification ignores the issue, thus providing no guidance.

The case of interest is that where an exception is thrown by some thread within a parallel construct, but not caught inside it. The most natural behaviour would be for parallel execution to terminate immediately, and the exception to be thrown on in the enclosing serial region by the master thread. In practice, though, the desired behaviour proves very difficult to implement. It is necessary that the thread throwing the exception has some way of interrupting the master thread. Unfortunately, no mechanism is provided in the Java language for interrupting a running thread. The `Thread.interrupt()` method only actually interrupts if the target thread is waiting. If it is running, it merely sets a flag.

Even more complex issues arise when an exception is thrown by one thread within a synchronisation or work-sharing construct, and caught outside this construct but *inside* the dynamically enclosing parallel region.

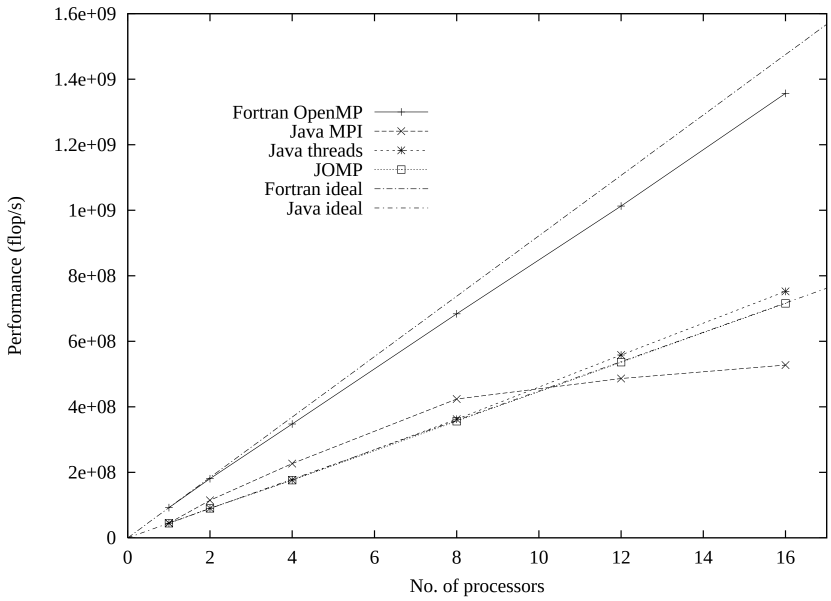

Figure 5: Performance of CFD application on Sun HPC 6500

6.5 Task based parallelism

OpenMP does not provide much support for task based parallelism: this shortcoming was noted in [17], and a solution proposed in the form of `task` and `taskq` directives. These provide a compact but powerful extension to OpenMP (though some implementation difficulties are noted), allowing parallelism over while loops, in recursive methods, and over complex data structures such as trees and lists, to be readily exploited. Since such parallelism is likely to be common in Java programs, a similar extension should be considered for JOMP.

7. 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.

8. 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. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, Lund, Sweden, Sept. 1999, pp. 99–105.

[3] 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.

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

[5] M. E. Kambites. Java OpenMP: Demonstration implementation of a compiler for a subset of OpenMP for Java, EPCC Technical Report EPCC-SS99-05, September 1999, available from `www.epcc.ed.ac.uk/ssp/1999/ProjectSummary/kambites.html`.

[6] International Organization for Standardization (ISO). Portable operating system interface (POSIX)—Part 1: system application program interface. ISO/IEC Standard 9945-1, 1996.

[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] J. Obdržálek. OpenMP for Java, EPCC Technical Report EPCC-SS-2000-08, September 2000, available from `www.epcc.ed.ac.uk/ssp/2000/ProjectSummary/obdrzalek.html`.

[11] J. Obdržálek and J. M. Bull. JOMP Application Program Interface. Available from `www.epcc.ed.ac.uk/research/jomp/pubs.html`, August 2000.

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

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

[14] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 2.0. Available from `www.openmp.org`, November 2000.

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

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

[17] S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible Control Structures for Parallelism in OpenMP. In *Proceedings of First European Workshop on OpenMP, Lund, Sweden*, pages 99–105, Spetember 1999.

[18] 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.

[19] 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.