JOMP—an OpenMP-like Interface for Java

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

M. E. Kambites
Department of Mathematics, University of York,
Heslington, York YO10 5DD, England, U.K.
mek100@york.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, JOMP, consisting of a compiler and a runtime library, both written entirely in Java, is presented, which implements a significant subset of the proposed specification.

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 [9] and C/C++ [8], and provides a higher level of abstraction to the programmer than, for example, programming with POSIX threads [4].

It is, of course, possible to write shared memory parallel programs using Java's *native threads* model [5, 7]. 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, that it would 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. 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, therfore, 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 [10], HPJava [2], Titanium [14] and SPAR Java [13]. 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 is written in Java, using the JavaCC compiler building system. In Section 5, we see an example of the JOMP system in operation. 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 is heavily based on the existing OpenMP standard for C/C++ [8], and 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 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 block or statement. 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. This facilitates the writing of portable code without the need to supply dummy library routines.

2.3 The `parallel` construct

Parallelism in a JOMP program is initiated by a `parallel` directive. A `parallel` directive takes the form:

```
//omp parallel [if(<cond>)]
//omp  [default (shared|none)] [shared(<vars>)]
//omp  [private(<vars>)] [firstprivate(<vars>)]
//omp  [reduction(<operation>:<vars>)]
<Java 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 array or objects. Note that declaring an object or array (reference) to be `private` creates only a new, uninitialised *reference* for each thread—no actual objects or arrays are allocated.

Example: computing the sum of an 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

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

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

Example: Simple parallel loop.

```
//omp parallel shared(a,b)
  {
//omp for
     for (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 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. A `single` directive takes the form:

```
//omp single [nowait]
<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.

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.

Example: see Section 5.

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 much 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.

Example: see Section 5.

2.11 Nesting of Directives

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.

If a thread encounters a `parallel` directive while already within the dynamic scope of a parallel region, a new team is created to execute the new parallel region. By default, this team contains only the current thread. Some compilers may support *nested parallelism* which, if enabled by the `setNested()` library method (see Section 2.12) or the `jomp.nested` system property (see Section 2.14), may cause extra threads to be created to execute the current region.

2.12 Library Functions

JOMP provides a range user-accessible library functions, implemented as static members of the class `jomp.runtime.OMP`.

`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.

getNumProcs() returns the maximum number of processors that could be assigned to the program or, where this cannot be ascertained, zero.

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.

2.13 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.

2.14 Environment

Some options can be provided to the OMP library at runtime, in the form of Java system properties.

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.

2.15 Differences from C/C++ standard

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

- 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 variables used for synchronisation are declared as volatile, this should not be a problem. However, it is not clear what effect the ambiguities in the Java memory model specification noted in [11] affect this issue.
- 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 for directive has no private clause, so a variable which is shared within a parallel region cannot be made private in an enclosed for directive.

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.

3.1 Structure of the Library

As well as the user-accessible functions and locks specified in Sections 2.12 and 2.13, 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 Machine class contains platform-specific code, such as JNI calls required to set up the system for parallelism. Increasingly, as JVMs become mulitporcessor aware, this will no longer be required. 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 2.13. The latter is also employed by the library to implement the `critical` directive.

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()`, assuming that we are 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 2.14. 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 3.12) and all the thread-specific data, 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 set up to be *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 responsiblility 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 posiibility threads from tying up resources indefinitely, threads waiting at a barrier will eventually yield—see Section 3.7.

3.5 The Machine class

For some purposes, it is necessary to use the Java Native Interface to make system calls not accessible directly through Java. For example, the `getNumProcs()` routine can only be implemented, if at all, by a call to an appropriate system routine.

The `Machine` class is designed to encapsulate all machine-specific code, making it accessible through a single interface. Thus, to compile on different platforms requires only the insertion of the appropriate `Machine.java` file. A generic, pure Java version of `Machine.java` is provided, so that the system can be ported to any platform without changes being necessary. If this is used, the `getNumProcs()` function always returns zero.

3.6 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`.

3.7 Barriers

The `Barrier` class implements a simple, static 4-way tournament barrier [3] 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 `Yield()`ing to other threads. The number of iterations can be set by calling the `setMaxBusyIter()` method, and can be tuned for different systems.

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

3.8 Reductions

The `Reduction` class is used to implement the `reduction` clause. It provides methods for the different reductions on different types described in Section 2. 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.

3.9 Scheduling

3.9.1 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. This 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.

3.9.2 The `Ticketer` *class*

The `Ticketer` class is used to facilitate dynamic allocation of work to different threads. A ticketer operates either in *counter mode* (the default) or in *loop mode*.

In *counter mode*, the synchronized `issue()` method is used to issue tickets. Successive calls to the `issue()` method return integer tickets, starting at zero. This facility is used to implement the `single` and `sections` constructs.

The first call to `issueBlock()` or `issueGuided()` switches the ticketer to *loop mode*. Calls to the `issueBlock()` and `issueGuided()` methods issue successive chunks of a loop, using a block and a guided scheduling strategy respectively.

Only one of the three issuing methods may meaningfully be used with each instance of the class `Ticketer`. They are implemented in a single class, to reduce the number of references that must be maintained by the library during execution.

The `resetTicketer()` 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.

3.9.3 Scheduling Support

The `OMP` class maintains for each thread a reference to a `Ticketer`. The `getTicket()`, `getLoopGuided()` and `getLoopBlock()` methods use the thread's `Ticketer` to return tickets and loop chunks as appropriate. The `resetTicket()` method advances the thread's reference to point to the next `Ticketer`. When all threads have advanced past a `Ticketer`, no reference to the object remains, and so it will be available for garbage collection.

The `getLoopStatic()` method is implemented directly in the `OMP` class without use of the ticketer. It is the only function which uses the secondary step field in the loop counter, and it returns the entire work allocation for each thread. This function maintains no internal state, so is reliant on the caller respecting the `isLast` flag and not trying to request another chunk.

The `getLoopRuntime()` function has the same effect as `getLoopStatic()`, `getLoopGuided()` or `getLoopBlock()`, depending on the user-specified runtime scheduling strategy.

The `setChunkBlock()`, `setChunkGuided()` and `setChunkRuntime()` methods are used to set a chunk size for use during scheduling, when none is provided by the user. The first two methods use sensible defaults, while the latter uses the user-specified size if available, or a sensible default otherwise.

3.10 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 to an Orderer. 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.

3.11 Locks

The Lock and NestLock classes described in Section 2.13 are implemented in a straightforward manner, using the Java synchronized method modifier to provide mutual exclusion.

3.12 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 NestLock. The getLockByName() method returns a reference to the lock associated with a given name, creating it and adding it to the hash table if necessary. Thus, we can think of the table as containing a lock for every possible name.

In parallel mode, the public startCritical() and stopCritical() methods get the appropriate lock, and attempt to set it and release it, respectively. In serial mode, both functions simply return with no effect.

4. THE JOMP COMPILER

In this section, we describe a simple compiler which implements a large subset of the specification suggested above. Currently, a few parts of the specification have yet to be implemented, such as nested parallelism, the default clause and reductions other than for + and ∗.

4.1 Basic Structure

The JOMP Compiler is built around a Java 1.1 parser provided as an example with the JavaCC [6] 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 (with the trivial exception of the Machine class), 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 The Symbol Table

The compiler needs to keep track of the types of local variables which are in scope. This is accomplished by means of a SymbolTable class, an instance of which is available as a static member of the OMPVisitor class. Conceptually, a symbol table is a stack of *scopes*, each of which contains a set of zero or more *entries*. Entries within a scope are uniquely identified by their *names*, and also contain fields for further information, such as variable type signatures. The symbol table has four operations. A new scope can be *created* on the top of the stack. Entries can be *added* to the uppermost scope on the stack. Entries can be *retrieved* by name, from the uppermost scope in the stack which contains that name. The uppermost scope can be *deleted*, removing from the table all entries added since the last scope was created.

Maintaining the symbol table requires overriding the visitors relating to several nonterminals. The MethodDeclaration, Block and ForStatement visitors are overridden to create new scopes to hold method parameters, locally declared variables and loop counters respectively. The LocalVariableDeclaration and FormalParameter visitors are overridden to add names to the table.

4.3 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.

4.4 The parallel directive

Upon encountering a parallel directive within a method, the compiler creates a new inner class, within the class containing the current method. If the method containing the parallel directive is static then the inner class is also static.

For each variable declared to be shared, the inner class contains a field of the same type signature and name. For each variable declared to be `firstprivate`, the inner class contains a field of the same type signature, named `__omp_fptemp_<varname>`. For each variable with a `reduction` operation specified, the inner class contains a field of the same type signature, named `__omp_lptemp_<varname>`.

The inner class has a single method, called `go`, which takes a parameter indicating an absolute thread identifier. For each variable declared to be `private` or `firstprivate`, the `go()` method declares a local variable with the same name and type signature. `firstprivate` variables are initialised from the corresponding field in the containing inner class, while `private` variables are uninitialised.

The main body of the `go()` method is the code to be executed in parallel. It is not necessary to make any changes to this block, other than to implement such work-sharing directives as may be found within it. The use of an inner class, and the declaration of appropriate local and class variables, effectively recreates the naming environment in which the code was originally located, so no modification is required to variable names. Finally, there is some code to perform any reductions, and to copy the resulting values into the appropriate class fields.

In place of the parallel construct itself, code is inserted to declare a new instance of the inner class, and to initialise the fields within it from the appropriate local 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.

```
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

4.5 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.

```
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

4.6 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, when 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);
```

4.7 The `critical` directive

Upon encountering a `critical` directive, the compiler inserts a call to `startCritical()` before the relevant block, and a call to `stopCritical()` after the block.

```
jomp.runtime.OMP.startCritical("name");
<block>
jomp.runtime.OMP.stopCritical("name");
```

4.8 The `barrier` directive.

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

4.9 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>
    }
```

4.10 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) {
        <code block>
    }
    jomp.runtime.OMP.resetTicket();
    [jomp.runtime.OMP.doBarrier();]
```

4.11 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();]
```

5. JOMP IN PRACTICE

JOMP has been applied to the Java Grande Forum MonteCarlo Benchmark [1]. This is a financial simulation, using Monte Carlo sampling techniques, which exhibits coarse grain parallelism.

The main loop of the program consists of 10000 iterations. Each iteration consists of a large calculation, capable of concurrent execution, and the writing of the resulting data, which must not overlap (but need not be ordered). The loop before parallelisation is:

```
  results = new Vector(nRunsMC);
  // Now do the computation.
  PriceStock ps;
  for( int iRun=0; iRun < nRunsMC; iRun++ ) {
    ps = new PriceStock();
    ps.setInitAllTasks(initAllTasks);
    ps.setTask(tasks.elementAt(iRun));
    ps.run();
    results.addElement(ps.getResult());
  }
```

The following changes instruct JOMP to parallelise the main loop, while ensuring that the `addElement()` method of the `results` vector is not called by more than one thread at once. The reference `ps` is declared to be private, since each thread will need its own copy. The reference `results` is a class field rather than a local variable, and so is shared by default.

```
  results = new Vector(nRunsMC);
  // Now do the computation.
  PriceStock ps;
  //omp parallel for private(ps) schedule(static)
  for( int iRun=0; iRun < nRunsMC; iRun++ ) {
    ps = new PriceStock();
    ps.setInitAllTasks(initAllTasks);
    ps.setTask(tasks.elementAt(iRun));
    ps.run();
    //omp critical
    {
        results.addElement(ps.getResult());
    }
  }
```

When tested on a Sun E3500/8 UltraSPARC, the original serial code took 80.46 seconds, the parallel code on one processor took 81.02 seconds, and the parallel code on all eight processors took 12.23 seconds. This represents a speedup factor of 6.58, and an efficiency of 82.2%. Similar results were obtained when the same code was parallelised by hand.

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 Data scope attributes

The major outstanding issue in the API design is to determine which kinds of variables should be allowed to appear in scope attribute clauses (e.g. `shared`, `private`, `reduction`). In the current implementation only local variables may appear in these clauses—all other variables are shared by default. This is overly restrictive, and it might be desirable to extend this to include, for example, fields of `this`. Some types of variable, on the other hand, such as class fields, probably should not be allowed in `private` clauses.

6.2 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. If an exception thrown from within the dynamic extent of a parallel region, but not caught within 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.

This has been attempted in the JOMP preprocessor and library. The `throws` clause on the `parallel` directive is used to specify classes of exception which may be thrown from within the dynamic extent of the parallel construct, but not caught inside it. 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.

6.3 Task based parallelism

OpenMP does not provide much support for task based parallelism: this shortcoming was noted in [12], and a solution proposed in the form of `task` and `taskq` directives. These provide a compact but powerful extension to OpenMP, 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. The system has been demonstrated in action on a simple parallel code.

Nevertheless, much more remains to be done. A complete specification is required, taking particular care with scoping issues. Although performance was taken into consideration in the design of the runtime library, it would undoubtedly benefit from further analysis and optimisation. It may also be of interest to explore the possibilities of designing a cleaner interface to the runtime library with the intention of it being used directly by the programmer. The compiler should be extended to implement the whole specification. Finally, the issues raised in Section 6 should be addressed.

8. REFERENCES

[1] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Methodology for Benchmarking Java Grande Applications. In *Proceedings of ACM 1999 Java Grande Conference*, pages 81–88. ACM Press, June 1999.

[2] 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.

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

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

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

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

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.