EPCC-SS99-05

Java OpenMP

Mark Kambites

Abstract

OpenMP is a specification of directives and library routines for shared memory parallel programming. At the time of writing, OpenMP standards exist for the C, C++ and Fortran programming languages.

This report investigates the definition and implementation of OpenMP for Java, based on Java's native threads model. A specification for OpenMP for Java is proposed and discussed. A compiler and runtime library, both written entirely in Java, are presented, which together implement a large subset of the proposed specification.

Contents

1 Introduction and Background

1.1 OpenMP Essentials

The OpenMP Application Program Interface is a standard for user-directed, shared-memory parallel programming. At the time of writing, OpenMP standards exist for C and C++ [3], and for Fortran [2].

The OpenMP programmer supplements his code with *directives*, which instruct an OpenMP-aware compiler to take certain actions. Some directives indicate pieces of code to be executed in parallel by a team of threads. Others indicate pieces of work capable of concurrent execution. Yet others provide synchronisation constructs, such as barriers and critical regions.

OpenMP is unusual among such systems in that directives may be *orphaned* — work-sharing and synchronisation directives may appear in functions which are capable of being called from within either parallel or serial regions, and must bind to the appropriate enclosing constructs at runtime.

The directives are specified in such a way that they will be ignored by a compiler without OpenMP support. This makes it easy to write portable code which exploits parallelism where available but runs sequentially where necessary.

In practice, OpenMP is sometimes implemented not directly by the compiler, but rather by a preprocessor. Such a preprocessor transforms the OpenMP directives into native constructs of the language, employing library and system calls as appropriate to provide parallelism. Our intention is to implement such a preprocessor for Java.

1.2 Java Native Threads Essentials

Java supports parallelism through its *native threads* model [4, 9, 10].

A thread may be created by declaring an instance of the library class `java.lang.Thread`, and started by calling its `start()` method. The thread's constructor takes as a parameter an object which implements the `Runnable` interface, the `run()` method of which is executed by the new thread. Alternatively, the `Thread` class may be extended to implement the `Runnable` interface itself, in which case its own `run()` method is used. A thread runs until it finishes its task, and any thread may wait for another thread to terminate, using the `join()` method.

1.3 OpenMP for Java

This report investigates the possibility of defining and implementing OpenMP for Java, using Java's native threads model. Section 2 suggests a possible OpenMP specification for Java, while Section 3 explains the differences from the existing standards.

Section 4 sees the introduction of a runtime library, written entirely in Java, which provides support for parallellism of the kind required by OpenMP. Section 5 introduces the JOMP preprocessor, which transforms Java code with OpenMP directives into Java with calls to the runtime library.

In Section 6, we see an example of the JOMP system in operation. Section 7 raises some outstanding issues, which would benefit from further research, while Section 8 concludes, evaluating work done.

2 OpenMP for Java

In this section, an informal specification is suggested for a Java implementation of OpenMP. This is heavily based on the existing OpenMP standards for other languages, but the exposition assumes no prior knowledge of OpenMP.

2.1 Format of Directives

The Java language has no standard form for compiler-specific directives, so we follow the Fortran standard by embedding directives in comments. A directive takes the form:

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

Directives are case sensitive. Some directives stand alone, as statements, while others act upon the immediately following Java block or statement. Directives do *not* need to be terminated with a line break, but failure to do so may of course affect the meaning of code to a non-OpenMP compiler.

Syntactically, a directive is a production of the Java `Statement` nonterminal, and so can appear anywhere a statement can appear. Those directives which act upon the immediately following blocks, incorporate those blocks. Semantically, it only makes sense to use a directive within a method body. In particular, for simplicity of implementation, directives may *not* appear within the body of a static class initialiser.

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 OpenMP-aware compiler.

2.3 The `parallel` construct

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

```
//omp parallel [if(<cond>)]
//omp          [shared(<vars>)] [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 `shared`, `private`, `firstprivate` and `reduction` clauses take as parameters comma-separated lists of variable names. Each *local* variable which is declared outside the code block but referenced within it must be specified to be `private`, `firstprivate` or `shared`. If a variable is `shared`, the same copy is in scope for each thread. If a variable is `private`, each thread receives its own, uninitialised copy. If it is `firstprivate`, each thread receives its own copy, initialised to the current value of the global copy.

Class members are always shared, and may not appear in a `shared`, `private`, `firstprivate` or `reduction` clause.

As always when programming in Java, it is vital to remember that all object and array names are just references. Thus, 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. Further, making an object `firstprivate` gives each thread its own *reference* to a *single* object.

If one or more `reduction` clauses are specified, the values of the appropriate variables held by each thread are combined as described in Section 2.11.

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])]
<for loop>
```

The loop must have a particular "canonical" form:

```
for(<integer-type> <var>; <var> <relation> <expr>; <inc>) ....
```

where

<integer-type> is one of short, int and long,

<relation> is one of <, <=, >=, >,

<inc> is one of <var>++, <var>--, ++<var>, --<var>, <var>+=<expr>, <var>-=<expr>,

<expr> is an expression of (or capable of implicit casting to) the same type as <var>, the value of which is loop invariant.

If a for directive is encountered within the dynamic extent of a parallel region, then the iterations of the loop will be divided amongst the threads in the current team, according to some *scheduling strategy* (see Section 2.5).

It is necessary that a given for construct is encountered by all threads in a particular team, or by none. The increment and comparison expressions are evaluated an unspecified number of times and without synchronisation, so side-effects may have unpredictable consequences.

If a for directive is encountered during serial execution, the loop will be executed by the single thread.

By default, there is an implicit barrier at the end of a for construct. This can be disabled by the use of the nowait clause. Note that there is no implicit barrier at the beginning of a for construct — execution of parts of the loop may commence before all threads have completed the previous operations.

If one or more reduction clauses are specified, the values of the appropriate variables held by each thread are combined as described in Section 2.11.

It is for the user to ensure that execution of the loop has the same (or an acceptably similar) effect when the order of iterations is altered, and when iterations are executed concurrently.

The ordered directive and clause are provided to assist with this task. The ordered clause is used to specify that an ordered block may appear within the loop body. 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>
```

Upon encountering the ordered directive, a thread will pause until the associated code block has been fully executed by all "previous" iterations, before itself commencing execution of the block. The `ordered` directive must appear within the *static* extent of the loop body, and only one such construct may appear within each loop. The ordered construct must be encountered exactly once during every iteration of the loop, or during none. All scheduling strategies ensure that each thread executes those iterations allocated to it in the correct order, thus ensuring that the `ordered` construct cannot deadlock.

2.5 Scheduling

A *scheduling strategy* dictates the manner in which the iterations of a loop are divided up between threads. Three different scheduling strategies are provided — `static`, `dynamic` and `guided`. The chunk size specified is interpreted in different ways for different strategies.

The scheduling strategies provided are the same as those documented in the existing OpenMP standards. For a more detailed explanation of their operation and intended use, please refer to either the C and C++ [3] or the Fortran [2] standards.

2.6 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 an "earlier" section without fear of deadlock. Also, placing long sections first may improve execution time.

As with a `for` construct, there is an implicit barrier at the end of a `sections` block unless the `nowait` clause is specified. There is no barrier implied at the beginning of a `sections` block.

2.7 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. A `single` block must be encountered by all threads in the team, or by none.

As with a `for` construct, there is an implicit barrier at the end of a `single` block unless the `nowait` clause is specified. There is no barrier implied at the beginning of a `single` block.

2.8 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>
```

There is no implied barrier at either the beginning or the end of a `master` construct, and there are no restrictions on which threads may or may not encounter the construct.

2.9 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 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.

Critical blocks are implemented using nested locks (see Section 2.15), so that a thread may safely enter a critical region while already within one of the same name.

2.10 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
```

All or none of the threads in a team much reach the barrier if the system is not to deadlock.

2.11 Reductions

A *reduction* uses an associative operation to combine the different values taken by a given private variable across different threads. Reductions are specified as clauses on either a `parallel` directive (see Section 2.3) or a `for` directive (see Section 2.4). The reduction will be carried out on completion of the appropriate construct. On a `for` directive, the result of the reduction will be copied to all threads.

For a reduction operation to make sense, the reduction variable must be `private`. On a `for` directive, the reduction variable must be `private` within the dynamically enclosing parallel region. The value resulting from the reduction will be copied to all threads. On a `parallel`, `parallel for` or `parallel sections` directive, the result of the reduction will be placed in the variable when execution of the parallel region finishes.

Currently supported are + and `*` reductions on the integer (`short`, `int` and `long`) and floating point (`float` and `double`) types, and `&&` and `||` reductions on the `boolean` type.

2.12 Combined parallel 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. It takes the form:

```
//omp parallel for <clauses>
<for loop>
```

Similarly, the `parallel sections` directive defines a parallel region containing only a single `sections` construct:

```
//omp parallel sections
{
    //omp section
    <code block>

    [//omp section
     <code block>]...
}
```

Almost all clauses which can be used with either a `parallel` directive or the appropriate load-sharing directive may be used with a combined work-sharing directive. The single exception is the `nowait` clause, which would have no purpose since the end of a parallel region always entails a barrier.

2.13 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 2.14) or the `jomp.nested` system property (see Section 2.16), may cause extra threads to be created to execute the current region.

2.14 Library Functions

Java OpenMP 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 current threads, 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`.

2.15 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.

2.16 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.

3 Comparison with Existing Standards

This section explains and attempts to justify the differences between our proposed Java OpenMP standard, and those already defined for C, C++ and Fortran.

3.1 The `default` clause

The `default` clause is not supported. Instead, behaviour is always as it is when `default(none)` is specified in the existing standards.

This decision has been taken for reasons of both efficiency and policy. The operation of the preprocessor is such that an overhead is incurred for each variable made either shared or private. Thus, it makes sense to minimise the number of variables which take either status.

Further, in the author's experience, many of the problems encountered by programmers attempting to parallelise code with OpenMP stem from a failure to give sufficient consideration to which variables should be private and which shared. The absence of the `default` clause may encourage users to give more thought to this issue.

3.2 Scoping and Loops

Perhaps the most significant difference is that `for` and `sections` constructs can have no effect on variable scope — all variables retain the status as `private` or `shared` given to them within the dynamically enclosing parallel region. This is primarily to allow an easy and efficient implementation, since code can be placed inline, with no need to declare extra variables.

For a `for` construct, we insist that the loop counter be declared in the loop initialiser. This ensures that each thread has its own copy of the counter, and makes it easy for the compiler to determine the type of the counter.

The `lastprivate` clause is not supported, because a variable cannot be private within a `for` construct and shared outside it. Should this prove too much of a handicap, it would be possible to support it only for variables already declared private, on the combined `parallel for` directive, or by copying the value from the final thread to all the other threads.

3.3 Reductions

Since the `for` directive cannot affect variable scopes, the requirement that the reduction variable for a `for` construct be `shared` within the dynamically enclosing parallel region is dropped. Instead, the reduction variable must be `private` and the reduced value will be copied to all threads.

3.4 Flushing

The `flush` directive is not supported. Nor is it necessarily guaranteed that variables will be flushed on other operations. See Section 7.2 for more discussion of this issue.

3.5 The `atomic` directive

The `atomic` directive is not supported. In the author's opinion, this directive is misleadingly named, and in practice causes many more problems than it solves. In any case, the kind of optimisations which the directive is designed to facilitate are unlikely to be possible in Java.

Should one wish to include support, it would be quite simple and efficient to implement where the atomically updated entity is an object or array, using Java's `synchronized` statement. However, in practice, the value is much more likely to be of a primitive type, in which case there is no obvious way to implement it short of using a single lock for all `atomic` statements.

3.6 The `threadprivate` directive and `copyin` clause

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 attempting to make these local would be a theoretical travesty and an implementational nightmare.

3.7 Library functions

Java OpenMP provides the same range of user-accessible library functions as specified in the C and Fortran standards. The functions are implemented as static members of the class `jomp.runtime.OMP` and the names have been changed to follow Java naming conventions. For example `omp_get_num_threads()` becomes `jomp.runtime.OMP.getNumThreads()`.

4 The JOMP Runtime Library

In this section and the next, we introduce a simple implementation of the specification proposed in Section 2. The implementation has two components. The *runtime library*, described in this chapter, provides constructs to support OpenMP parallelism in terms of Java's native threads model. The *preprocessor*, described in the following chapter, transforms Java with OpenMP directives into Java with appropriate calls to the library.

4.1 Structure of the Library

As well as the user-accessible functions and locks specified in Sections 2.14 and 2.15, 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 2.14, 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, and are described in Section 4.4.

The `Machine` class contains platform-specific code, such as JNI calls required to set up the system for parallelism. It is described in Section 4.5.

The `Barrier` class implements a barrier, and is used for internal thread-management purposes, as well as for implementing the numerous OpenMP constructs which require such a device. It is described in Section 4.7.

The `Orderer` class is used to facilitate implementation of the OpenMP `ordered` construct, and is described in Section 4.10. The `Reducer` class implements reductions of variables, and is described in Section 4.8.

The `Ticketer` and `LoopData` classes are used to facilitate scheduling, and are described in detail in Section 4.9.

The `Lock` and `NestLock` classes implement the user-accessible locks described in Section 2.15. The latter is also employed by the library to implement the OpenMP `critical` directive.

4.2 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 1.1) 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 sheer 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 seems to be 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. See Section 5.3 for a discussion of how these methods are used.

4.3 Initialisation

Initialisation is divided into two parts.

The static initialisation for the class `jomp.runtime.OMP` reads the system properties documented in Section 2.16. These are used to set up the numbers of threads and of processors to use, and to set up the static subclass `Options`, which contains configuration information.

The `start()` method is called on demand, when parallelism is invoked. It initialises the critical region table (see Section 4.12) and all the thread-specific data, creates a team of threads, and sets them running, whereupon they wait to be assigned a task.

4.4 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 is 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 anything useful.

4.5 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. Further, on some systems it is necessary to make system calls to configure parallelism appropriately. For example, under Solaris one must call the `setconcurrency()` routine to ensure that each Java thread can run on its own processor.

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.

4.6 Nested Parallelism

Nested parallelism is not currently supported. 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 1, and the task is executed. Finally, the original values of the thread-specific data are restored.

The `setNested()` method does nothing, and the `getNested()` method always returns `false`.

4.7 Barriers

The `Barrier` class implements a simple, static 4-way tournament barrier [7] for an arbitrary number of threads. The 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 cooperative rather and pre-emptive multitasking. If for some reason the threads are not each allocated their own processor, busy-waiting can cause deadlock. To counteract 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. Different values are likely to work best with 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.

4.8 Reductions

The `Reduction` class is used to implement the OpenMP `reduction` construct. It provides methods for the different reductions on different types described in Section 2.11.

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 to all threads 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`. In fact, the amount of synchronisation employed in the current implementation is excessive. Significant time savings might be possible with improvements in the efficiency of the implementation.

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.

4.9 Scheduling

4.9.1 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.

The `LoopData` class and associated routines are all implemented using loop counters of type `long`. Loop counters of other types are cast to type `long`. For efficiency reasons, it might be worth implementing separate routines for each possible counter type.

4.9.2 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 OpenMP `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.

4.9.3 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.

4.10 Ordering Support

The Orderer class is used to implement the OpenMP 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 startOrderer() 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.

4.11 Locks

The Lock and NestLock classes described in Section 2.15 are implemented in a straightforward manner, using the Java synchronized method modifier to provide mutual exclusion.

4.12 Critical Regions

The requirement that names of critical regions be global in scope presents a problem. OpenMP 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.

5 The JOMP Compiler

In this section, we describe a simple compiler which implements a large subset of the OpenMP specification suggested above.

5.1 Mode of Operation

The JOMP Compiler is built around a Java 1.1 parser provided as an example with the JavaCC [1] 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 of the methods which unparse particular nonterminals.

These overriding methods output modified code, which includes calls to the runtime library to implement appropriate parallelism.

5.2 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.

5.3 Personal Identity Revisited

As discussed in Section 4.2, 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.

5.4 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 environement 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.

5.5 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.

5.6 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);
```

5.7 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");
```

5.8 The `barrier` directive.

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

5.9 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>
}
```

5.10 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();]
```

5.11 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();]
```

6 JOMP in Practice

The time available for the project has not permitted extensive testing, but JOMP has been applied to the Java Grande Forum Monte Carlo Benchmark. This is a financial simulation, using Monte Carlo sampling techniques.

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 calls 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 automatically shared.

```
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 [8].

7 Outstanding Issues

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

7.1 Exception Handling

Exceptions are an important feature of the Java language, and it is worth considering how they will be handled by an OpenMP implementation. Exceptions are present in C++, but they are less widely used than in Java and the C++ OpenMP 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.

7.2 Flush and the Java Memory Model

The Java memory model specification [6] is very complex. At the time of writing there are some doubts about whether it says what the authors intended, and whether it is correctly implemented by the majority of existing compilers. [11]

For these reasons, and for want of time, I have refrained from considering in detail the memory model. In particular, I have not implemented the `flush` directive, or given consideration to whether there need to be implicit flush operations after or during certain constructs.

At some point, preferably when the issues raised by [11] have been satisfactorily resolved, more investigation of this matter would be helpful.

7.3 Error Handling

The current JOMP preprocessor has no error handling worth speaking of. Many directive errors and virtually all errors in the underlying code cause an exit with a stack dump. In practice, it is necessary to ensure that a program compiles correctly with the sequential compiler before attempting to run the JOMP preprocessor on it.

7.4 Efficiency Issues

While some thought has been given to the efficiency of the mechanisms used in the runtime library and the code generated by the preprocessor, the time available has not permitted extensive comparison of alternative approaches. Significant savings could almost certainly be made by improvements in this area.

8 Conclusion

A possible specification for Java OpenMP has been presented, and some of the issues surrounding it discussed. A preprocessor and library have been implemented which together utilise Java's native threads model to implement a large subset of the proposed specification.

Some issues still require further consideration. In particular, the handling of exceptions and the implications of the Java memory model will need to be examined in more depth. However, it is hoped that with the resolution of these issues, Java OpenMP will prove a useful tool for programmers of high performance computers.

Acknowledgements

Thanks are due to my supervisor, Mark Bull, and to the other staff at EPCC. I am also indebted to my fellow SSP students — Alexander, Alexandros, Benoit, Daniel, Jo, Klaus, Mark, Robert, Tom and Wing-yun — for helping in ways too numerous to mention.

References

[1] JavaCC page. http://www.suntest.com/JavaCC/.

[2] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface*, October 1997.

[3] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, October 1998.

[4] Thomas W. Christopher and George K. Thiruvathukal. Introduction to high-performance Java computing, June 1999. Teaching slides.

[5] David Flanagan. *Java in a Nutshell*. O'Reilly, 1997.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*, chapter 17. Addison-Wesley, 1996.

[7] Dirk Grunwald and Suvas Vajracharya. Efficient barriers for distributed shared memory computers. *Univ. Colorado Technical Report CU-CS-703-94-93*, September 1993.

[8] Alexandros Karatzoglou. Developing a parallel benchmarking suite for Java Grande applications. SSP project report, Edinburgh Parallel Computing Centre, 1999.

[9] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1997.

[10] Scott Oaks and Henry Wong. *Java Threads*. O'Reilly, 1997.

[11] W. Pugh. Fixing the Java memory model. pages 89–98. ACM Java Grande Conference, June 1999.

Mark Kambites is currently studying for an Masters of Mathematics degree in Mathematics and Computer Science, at the University of York.

This project was supervised by Dr. Mark Bull.