JOMP Application Program Interface
Version 0.1 (draft)

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

Mark Bull
EPCC, University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ,
Scotland, U.K.
email: `markb@epcc.ed.ac.uk`

August 25, 2000

Contents

Chapter 1

Introduction

This document specifies an API for shared memory parallel programming in Java, consisting of compiler directives, a run-time class library, and system properties. This functionality is collectively known as the JOMP API.

This document bears a close relationship to the OpenMP C/C++ Application Program Interface [1], but this does not imply any endorsement of the JOMP API by the OpenMP Architecture Review Board.

1.1 Definition of Terms

The following terms are used in this document:

barrier	A synchronisation point that must be reached by all threads in a team (if at all). Each thread waits until all the threads have arrived at this point. There are both explicit barriers identified by barrier directives, and implied barriers associated with other directives.
construct	A construct is a statement. It consists of a JOMP directive and the subsequent structured block. Note that some directives are not part of a construct.
dynamic extent	All statement in the *lexical extent* a method which is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a region.
lexical extent	Statements lexically contained within a structured block.
master thread	The thread that creates a team when a parallel region is entered.
parallel region	A structured block executed by multiple threads in parallel.
private	Accessible by only one thread in a team. This is unrelated to the `private` visibility modifier.
serial region	Statements outside of a parallel region, executed by the master thread only.

serialise	To execute a parallel construct with a single thread.
structured block	A structured block is a block that has a single entry and a single exit. A block is not structured if there is jump into or out of the block (for example by the use of `break` or `continue`, or by throwing an exception which is caught outside the block).
variable	According to the *Java Language Specification* [2] (§4.5), *a variable* is a storage location and has an associated type, that is either a primitive type, or a reference type. A variable of a primitive type holds a value of this type, whilst a variable of a reference type holds a reference to the value of this type. (There are three kind of reference types: class types, interface types, and array types). In this text, *variable* also refers to a name, or a qualified name, which denotes the storage location. The actual meaning should be clear from the context.

1.2 Execution Model

JOMP uses the fork-join model of parallel execution. A program written using the JOMP API begins execution on a single thread (called the *master thread*). The master thread executes in a serial region until the first parallel construct is encountered, whereupon the master thread creates a team of threads, which includes itself. Each thread then executes the code in the dynamic extent of the parallel region, except for blocks contained in work-sharing constructs. Work sharing constructs must be encountered by all the threads in team (or by none at all): the code inside these constructs is executed only once, but may be distributed across multiple threads. The implied barrier at the end of the the work-sharing construct without a `nowait` clause is executed by all the threads in the team.

*** There should be a paragraph here about when modifications to shared variables are guaranteed to be complete, but I'm not sure what it should say, given the current state of the Java memory model! ***

Upon completion of the parallel construct, the threads in the team synchronise at an implied barrier, and the master thread continues executing subsequent statements. Any number of parallel constructs may appear in a program, and thus there may be any number of forks and joins.

The JOMP API allows the use of *orphaned* directives, that is directives which appear in the dynamic extent of a parallel construct but not in its lexical extent.

Unsynchronised calls to Java output methods that write to the same file may result in non-deterministic ordering of output. Unsynchronised calls to Java input methods that read from the same file may result in non-deterministic ordering of input. Unsynchronised I/O where each thread accesses a different file will produce the same results as the serial execution of the I/O methods.

Chapter 2

Directives

Directives are special Java single line comments that are identified with a unique *sentinel*. The directive sentinel is structured do that the directives are treated as Java comments by Java compilers that do not support JOMP.

This section addresses the following topics:

- Directive format (see Section 2.1, page 6).
- The `only` construct (see Section 2.2, page 7).
- The `parallel` construct (see Section 2.3, page 7).
- Work-sharing construct (see Section 2.4, page 8).
- Combined parallel work-sharing constructs (see Section 2.5, page 12).
- Synchronisation constructs (see Section 2.6, page 13).
- Data environment clauses (see Section 2.7, page 14).
- Directive binding (see Section 2.8, page 18).
- Directive nesting (see Section 2.9, page 19).

2.1 Directive Format

The syntax of a JOMP directive is specified informally as follows:

```
//omp directive-name [clause[ clause] ...]
[//omp [ clause[clause] ...]
...
```

The `//omp` at the start of each line of the directive is the sentinel. Directives are case sensitive, and the order in which clauses appear is not significant. Clauses may be repeated as required, subject to the restrictions specified in the description of each clause. If a *list* appears in a clause, it must specify only variables. Only one *directive-name* is permitted per directive. A JOMP directive applies to at most one succeeding statement, which must be a structured block.

2.2 `only` Directive

The `only` directive permits simple conditional compilation. It takes the following form:

```
//omp only statement
```

This directive is replaced by *statement*, so that the statement is only executed if the a JOMP aware compiler is used, and is ignored by standard Java compilers. This facilitates the writing of code which will compile and run correctly using either a JOMP aware compiler or a standard Java compiler.

2.3 `parallel` Construct

The following directive defines a *parallel region*, which is a region of the program to be executed by multiple threads in parallel. This is the fundamental construct which enables parallel execution.

```
//omp parallel [clause[ clause] ...]
      structured-block
```

The *clause* is one of the following:

`if(`*boolean-expression*`)`

`private(`*list*`)`

`shared(`*list*`)`

`firstprivate(`*list*`)`

`default(shared | none)`

`reduction(`*operator*`:`*list*`)`

For information on the `private`, `shared`, `firstprivate`, `default` and `reduction` clauses, see Section 2.7, page 14. When a thread encounters a `parallel` construct, a team of threads is created if there is no `if` clause, or if the `if` expression evaluates to TRUE. If the value of the `if` expression is FALSE, the region is serialised.

The number of threads is constant during the execution of a parallel region. The number of threads may be changed between parallel regions either explicitly via the `OMP.setNumThreads` method, or automatically. Automatic adjustment can be enabled and disabled via the use of the `OMP.setDynamic` function (see Section 3.1.6, page 22) or the `jomp.dynamic` system property (see Section 4.3, page 27).

The statements contained within the dynamic scope of the parallel region are executed by each thread, and each thread can execute a path which is different from the other threads. Directives encountered outside the lexical extent of a parallel region but inside the dynamic extent are referred to as *orphaned* directives.

There is an implied barrier at the end of the parallel region. Only the master thread of the team continues execution after the end of the parallel region.

If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team and it becomes the master thread of that team. Nested parallel regions are serialised by default, in which case the nested region is executed by a team consisting of one thread. The default behaviour may be changed either by using the `OMP.setNested` function (see Section 3.1.8, page 23) or the `jomp.nested` system property (see Section 4.4, page 27).

> **Implementation note** *The* `OMP.setNested` *function and the* `jomp.nested` *system property currently have no effect: the default behaviour cannot be altered.*

Restrictions to the `parallel` directive are as follows:

- At most one `if` clause can appear on the directive.
- It is unspecified whether any side-effects inside the `if` expression occur.
- Any exception thrown inside a parallel region must be caught within the dynamic extent of the same structured block, and must be caught by the same thread that threw the exception.

2.4 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct. The sequence of work-sharing constructs and barriers encountered must be the same for all threads in a team.

JOMP defines the following work-sharing constructs:

- `for` directive (see Section 2.4.1, page 8)
- `sections` directive (see Section 2.4.2, page 11)
- `single` directive (see Section 2.4.3, page 11)
- `master` directive (see Section 2.4.4, page 12)

2.4.1 `for` Construct

The `for` directive specifies a loop whose iterations should be executed in parallel. The iterations of the `for` loop are distributed across the existing threads in the current team. The syntax of the `for` construct is as follows:

```
//omp for [clause[ clause] ...  ]
     for-loop
```

The *clause* is one of the following:

`private(`*list*`)`

`firstprivate(`*list*`)`

`lastprivate(`*list*`)`

`reduction(`*operator*`:` *list*`)`

`ordered`

`schedule(`*kind*`[,` *chunk_size*`])`

`nowait`

The `for` directive places restrictions on the structure of the corresponding `for` loop. Specifically, the corresponding `for` loop must have *canonical* shape:

`for (`*init-expr*`; var` *logical-op* `b;` *incr-expr*`)`

init-expr	One of the following: `var = lb` *integer-type* `var = lb`

incr-expr	One of the following:
	`++var`
	`var++`
	`--var`
	`var--`
	`var += incr`
	`var -= incr`
	`var = var + incr`
	`var = var - incr`

`var`	A variable of signed integer type (`byte`, `short`, `int` or `long`). This variable is implicitly made private for the duration of the `for` loop. Unless it is declared as `lastprivate`, it value after the loop is indeterminate.

logical-op	One of the following:
	`<`
	`<=`
	`>`
	`>=`

`lb, b` and `incr`	Loop invariant integer expressions. Any evaluated side-effects of these expressions produce indeterminate results.

Note that the canonical form allows the number of loop iterations to be computed on entry to the loop. This computation is performed with values of the same type as `var`. In particular, if the value of `b - lb + incr` cannot be represented in this type, the result is indeterminate.

The `schedule` clause specifies how the iterations of the `for` loop are divided among the threads of the team. The correctness of the program must not depend on which thread executes a particular iteration. The value of *chunk_size*, if specified, must be a loop-invariant integer expression with a positive value. There is no synchronisation during the evaluation of this expression, so any side effects produce indeterminate values. The schedule *kind* can be one of the following:

`static`	When `schedule(static,`*chunk_size*`)` is specified, iterations are divided into chunks of size *chunk_size*. The chunks are statically assigned to threads in the team in a round-robin fashion based on the thread number. When no chunksize is specified, the iteration space is divided into approximately equal chunks that minimise the maximum number of iterations performed by any thread.
`dynamic`	When `schedule(dynamic,`*chunk_size*`)` is specified, a chunk of *chunk_size* iterations is assigned to each thread. When it finishes its chunk, it is dynamically assigned the next available chunk, until no more remain. Note that the final chunk may contain fewer iterations. If *chunk_size* is not specified, its value defaults to 1.
`guided`	When `schedule(guided,`*chunk_size*`)` is specified, the iterations are also assigned dynamically, but the chunks have exponentially decreasing sizes. The size of each chunk is approximately the number of remaining iterations divided by twice the number of threads. The value of *chunk_size* determines the minimum number of iterations in a chunk, except possibly the last (which may contain fewer). If *chunk_size* is not specified, its value defaults to 1.
`runtime`	When `schedule(runtime)` is specified, the scheduling decision is deferred to run-time, and is determined by the value of the `jomp.schedule` system property, which is described in Section 4.1, page 26. If this property is not set, the schedule is implementation dependent. When `schedule(runtime)` is specified, *chunk_size* may not be specified in the `schedule` clause.

In the absence of a `schedule` clause, the default schedule is implementation dependent.

A JOMP program should not rely on the details of the schedule for correct execution, as the implementation may vary across different compilers.

The `ordered` clause must be present when `ordered` directives appear in the dynamic extent of the `for` construct.

There is an implicit barrier at the end of a `for` construct, unless a `nowait` clause is present.

Restrictions to the `for` directive are as follows:

- The `for` loop must be a structured block, and in addition, its execution must not be terminated by the `break` or `continue` statements.
- The values of the loop control expressions of the `for` loop must be the same for all threads in a team.
- The `for` loop iteration variable must have a signed integer type.
- Only one `schedule` clause can appear on a `for` directive.
- Only one `ordered` clause can appear on a `for` directive.
- Only one `nowait` clause can appear on a `for` directive.
- It is unspecified if any side-effect within the *chunk_size*, `lb`, `b` or `incr` expressions occurs.
- The value of the *chunk_size* expression must be the same for all threads in the team.

Implementation note *In the current implementation, default schedule is* `static`*, with no given chunk_size.*

2.4.2 `sections` Construct

The `sections` directive identifies a non-iterative work-sharing construct that specifies a set of constructs that are to be divided among threads in a team. The syntax of `sections` directive is as follows:

```
//omp sections [clause[ clause] ...]
     {
          //omp section
               structured-block
          [//omp section
               structured-block
          .
          .
          .]
     }
```

The *clause* is one of the following:

`private(`*list*`)`

`firstprivate(`*list*`)`

`lastprivate(`*list*`)`

`reduction(`*operator: list*`)`

`nowait`

For information on the `private`, `firstprivate`, `lastprivate` and `reduction` clauses, see Section 2.7, page 14.

Each section is preceded by a `section` directive. There is an implicit barrier at the end of a `sections` construct, unless a `nowait` clause is present.

Restrictions to the `sections` directive are as follows:

- A `section` directive may not appear outside the lexical extent of the `sections` directive.
- Only one `nowait` clause can appear on a `sections` directive.

2.4.3 `single` Construct

The `single` directive identifies a construct that specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The syntax of the single directive is as follows:

```
//omp single [clause[ clause] ...  ]
     structured-block
```

The *clause* is one of the following:

`private(`*list*`)`

`firstprivate(`*list*`)`

`nowait`

For information on the `private` and `firstprivate` clauses, see Section 2.7, page 14.

There is an implicit barrier at the end of a `single` construct, unless a `nowait` clause is present.

Restrictions to the `single` directive are as follows:

- Only one `nowait` clause can appear on a `single` directive.

2.4.4 `master` Construct

The `master` directive identifies a construct that specifies that the associated structured block is executed by the master thread in the team.

The syntax of the master directive is as follows:

```
//omp master
      structured-block
```

The other threads in the team do no execute the block. There is *no* barrier at the end of a `master` construct.

2.5 Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are short cuts for specifying a parallel region whose contents consist of a single work-sharing construct. The semantics of these directives are identical to specifying a `parallel` directive immediately followed by a single work-sharing construct.

The following sections describe the parallel work-sharing constructs:

- Section 2.5.1, page 12, describes the `parallel for` directive.
- Section 2.5.2, page 13, describes the `parallel sections` directive.

2.5.1 `parallel for` Construct

The `parallel for` directive is a shorthand for a `parallel` region containing a single `for` directive. The syntax of the `parallel for` directive is as follows:

```
//omp for [clause[ clause] ...  ]
      for-loop
```

This directive admits all the clauses of the `parallel` directive (see Section 2.3, page 7) and the `for` directive (see Section 2.4.1, page 8) with the exception of the `nowait` clause, with identical meanings and restrictions. The semantics are identical to specifying a `parallel` directive immediately followed by a `for` directive.

2.5.2 `parallel sections` Construct

The `parallel sections` directive is a shorthand for a `parallel` region containing a single `sections` directive. The syntax of the `parallel sections` directive is as follows:

```
//omp parallel sections [clause[ clause] ...]
     {
          //omp section
               structured-block
          [//omp section
               structured-block
          .
          .
          .]
     }
```

This directive admits all the clauses of the `parallel` directive (see Section 2.3, page 7) and the `sections` directive (see Section 2.4.2, page 11) with the exception of the `nowait` clause, with identical meanings and restrictions.

2.6 Synchronisation Constructs

The following sections describe the synchronisation constructs:

- Section 2.6.1, page 13, describes the `critical` directive.
- Section 2.6.2, page 14, describes the `barrier` directive.
- Section 2.6.3, page 14, describes the `ordered` directive.

2.6.1 `critical` Construct

The `critical` directive specifies a structured block which may be executed by only one thread at a time. The syntax of the `critical` directive is as follows:

```
//omp critical [(name)]
     structured-block
```

The optional *name* may be used to identify the critical region. These names comprise a separate namespace which is distinct from Java namespaces.

A thread waits at the beginning of a critical region until no other thread is executing any critical region with the same name. Unnamed critical regions are treated as having the same (null) name.

2.6.2 `barrier` Construct

The `barrier` directive synchronises all the threads in a team. When encountered, each thread waits until all the other threads in the team have reached this point. After all threads have encountered the barrier, each thread continues executing the subsequent statements. The syntax of the `barrier` directive is as follows:

```
//omp barrier
```

Restrictions to the `barrier` directive are as follows:

- The smallest statement containing a `barrier` directive must be a block.
- A `barrier` directive must be encountered either by all threads in a team, or by none of them.

2.6.3 `ordered` Construct

An `ordered` directive may appear in the dynamic extent of a `for` or `parallel for` construct that has an `ordered` clause. The `ordered` directive specifies a block which must be executed in the same order as the iterations of the loop would be executed if it were run sequentially. The syntax of the `ordered` directive is as follows:

```
//omp ordered
      structured-block
```

Restrictions to the `ordered` directive are as follows:

- An `ordered` directive may only appear in the dynamic extent of a `for` or `parallel for` construct that has an `ordered` clause.
- An iteration of an `ordered` loop must only encounter one `ordered` directive, and may not encounter it more than once.

2.7 Data Scope Attribute Clauses

Several directives accept clauses, that allow a user to control the scope attributes of variables for the duration of the region. Scope attribute clauses apply only to variables in the *lexical extent* of the directive on which clause appears. Not all of the following clauses are allowed on all directives. The list of clauses that are valid on a particular directive is described with the directive.

If a variable is visible when a parallel or work-sharing construct is encountered, and the variable is not specified in a scope attribute clause, then the variable is shared. Variables with automatic storage duration declared within the dynamic extent of a parallel region are private.

Most of the clauses accept a *list* argument, which is a comma-separated list of variables that are visible. Clauses may be repeated as needed, but no variable may be specified in more than one clause, except that a variable can be specified in both a `firstprivate` and a `lastprivate` clause.

The following sections describe the data scope attribute clauses:

- Section 2.7.1, page 15, describes the `private` clause.
- Section 2.7.2, page 16, describes the `firstprivate` clause.
- Section 2.7.3, page 16, describes the `lastprivate` clause.
- Section 2.7.4, page 16, describes the `shared` clause.
- Section 2.7.5, page 17, describes the `default` clause.
- Section 2.7.6, page 17, describes the `reduction` clause.

Implementation note *In the current implementation all variables, except the local variables and method parameters, should have an associated type provided by a user. So the syntax of* list *is as follows:*

```
list ::= TypeName_opt VariableName (, TypeName_opt VariableName)*
```

2.7.1 `private`

The `private` clause declares the variables in *list* to be private to each thread in a team. The syntax of the `private` clause is as follows:

```
private (list)
```

The behaviour of a variable specified in a `private` clause is as follows. A new variable of the same type with automatic storage duration is allocated within the associated structured block (thus, its visibility is defined by this block). This allocation occurs once for each thread in the team. If a variable holds a reference to a class instance object, a default constructor is invoked and the reference to the newly created object is assigned to the variable. Otherwise, the initial value is indeterminate. The original variable (object referenced by this variable) has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct, and has an indeterminate value upon exit from the construct.

In the lexical extent of the directive construct, the variable references the new private storage location allocated by the thread.

The restrictions to the `private` clause are as follows:

- A variable with a class type that is specified in the `private` clause must have an accessible default constructor.
- A variable specified in a `private` clause must not have been declared `final`.
- A variable specified in a `private` clause must not have an interface type.
- Variables that are private within a parallel region cannot be specified in a `private` clause on an enclosed work-sharing or `parallel` directive. As a result, variables that are specified `private` on a work-sharing or `parallel` directive must be shared in the enclosing parallel region.

2.7.2 `firstprivate`

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause. The syntax of the `firstprivate` clause is as follows:

```
firstprivate (list)
```

Variables specified in the *list* have `private` clause semantics described in Section 2.7.1, page 15, except that each new private variable is initialised as if there were an implied declaration inside the structured block, and the initialiser is the value of the original variable. If a variable has a class or an array type, the `clone()` method of the object referenced by the original variable is called to create a new instance of the class.

The restrictions to the `firstprivate` clause are as follows:

- All restrictions for `private` apply except for the restriction about default constructors.
- An object referenced by a variable that is specified as `firstprivate` must have an accessible `clone()` method.

Implementation note *In the current implementation, all variables specified to be* `firstprivate` *must be initialised prior to the directive, on which they are declared* `firstprivate`*.*

2.7.3 `lastprivate`

The `lastprivate` clause provides a superset of the functionality provided by the `private` clause. The syntax of the `lastprivate` clause is as follows:

```
lastprivate (list)
```

Variables specified in the *list* have `private` clause semantics, described in Section 2.7.1, page 15. When a `lastprivate` clause appears on the directive that identifies a work-sharing construct, the value of each `lastprivate` variable from the sequentially last iteration of the associated loop, or the lexically last `section` directive, is assigned to the original variable. Variables that are not assigned a value by the last iteration of the `for` or `parallel for`, or by the lexically last `section` of the `sections` or `parallel sections` directive, have indeterminate values after the construct.

The restrictions to the `lastprivate` clause are as follows:

- All restrictions for `private` apply.

2.7.4 `shared`

This clause shares variables that appear in the *list* among all the threads in a team. All threads within a team access the same storage area for `shared` variables. The syntax of the `shared` clause is as follows:

```
shared(list)
```

> **Implementation note** *In the current implementation,* `shared` *variables which are not initialised on the entry of the* `parallel` *construct, must be assigned a value during the execution of the* `parallel` *region.*

2.7.5 `default`

The `default` clause allows the user to affect the data scope attributes of variables. The syntax of the default clause is as follows:

```
default(shared | none)
```

Specifying `default(shared)` is equivalent to explicitly listing each currently visible variable in a `shared` clause. In the absence of an explicit `default` clause, the default behaviour is the same as if `default(shared)` was specified.

Specifying `default(none)` requires that each currently visible variable that is referenced in the lexical extent of the parallel construct must be explicitly listed in a data scope attribute clause, unless it is `final`, specified in an enclosed data scope attribute clause, or used as a loop control variable referenced only in a corresponding `for` or `parallel for`.

Only a single `default` clause may be specified on a `parallel` directive.

> **Implementation note** *When* `default(none)` *is specified, a variable not explicitly listed in a* `shared` *clause may not be specified on an enclosed data scope attribute clause.*

2.7.6 `reduction`

This clause performs a reduction on the primitive type variables that appear in the *list*, with the operator *op*. The syntax of the `reduction` clause is as follows:

```
reduction(op:list)
```

Where *op* is one of the following operators: +, $*$, -, &, ˆ, |, &&, or ||.

A private copy of each variable in the *list* is created, as if the `private` clause had been used. The private copy is initialised according to the operator (see the table below).

At the end of the region for which the `reduction` clause was specified, the shared variable is updated to reflect the result of combining the original value of the (shared) reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler may freely reassociate the computation of the final value. (The partial results of a subtraction reduction are added to form the final value.)

The value of the shared variable becomes indeterminate when the first thread reaches the containing clause and remains so until the reduction computation is complete. Normally, the computation will be complete at the end of the `reduction` construct; however, if the `reduction` clause is used on a construct to which `nowait` is also applied, the value of the shared variable remains indeterminate until a barrier synchronisation has been performed to ensure that all threads have completed the `reduction` clause.

The following table lists the operators that are valid, their initialisation values and the types permitted with a particular operator. The actual initialisation value will be consistent with the data type of the reduction variable.

Operator	Initialisation	Allowed types
+	0	byte, short, int, long, char, float, double
*	1	byte, short, int, long, char, float, double
-	0	byte, short, int, long, char, float, double
&	~0	byte, short, int, long, char
|	0	byte, short, int, long, char
^	0	byte, short, int, long, char
&&	`true`	boolean
||	`false`	boolean

The restrictions to the `reduction` clause are as follows:

- The type of the variables in the reduction clause must be valid for the `reduction` operator.
- A variable that is specified in the `reduction` clause must not be declared `final`.
- A variable that is specified in the `reduction` clause must be shared in the enclosing context.

> **Implementation note** *In the current implementation, all reductions are implemented using a barrier for performance reasons. As a result, value of the shared variable is available immediately after the end of the* `reduction` *construct.*

2.8 Directive Binding

Dynamic binding of directives must adhere to the following rules:

- The `for, sections, single, master, and barrier` directives bind to the dynamically enclosing `parallel`, if one exists. If no parallel region is currently being executed, the directives have no effect.
- The `ordered` directive binds to the dynamically enclosing `for`.
- The `critical` directive enforces exclusive access with respect to `critical` directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing `parallel`.

2.9 Directive Nesting

Dynamic nesting of directives must adhere to the following rules:

- A `parallel` directive dynamically inside another `parallel` logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.
- `for, sections,` and `single` directives that bind to the same `parallel` are not allowed to be nested inside each other.
- `critical` directives with the same name are not allowed to be nested inside each other.
- `for, sections,` and `single` directives are not permitted in the dynamic extent of `critical, ordered,` and `master` regions.
- `barrier` directives are not permitted in the dynamic extent of `for, ordered, sections, single, master,` and `critical` regions.
- `master` directives are not permitted in the dynamic extent of `for, sections,` and `single` directives.
- `ordered` directives are not allowed in the dynamic extent of `critical` regions.
- Any directive that is permitted when executed dynamically inside a `parallel` region is also permitted when executed outside a `parallel` region. When executed dynamically outside a user-specified `parallel` region, the directive is executed with respect to a team composed of only the master thread.

Chapter 3

Run-time Library Functions

This section describes the JOMP run-time library. This library is implemented by the following three classes. Functions that can be used to control and query the parallel execution environment are static methods of the `jomp.runtime.OMP` class. The `jomp.runtime.Lock` and `jomp.runtime.NestLock` classes are introduced to implement locks that can be used to synchronise access to data.

`jomp.runtime.Lock` is a class type capable of representing that a lock is available, or that a thread owns a lock. These lock are referred to as *simple locks*.

`jomp.runtime.NestLock` is a class type capable of representing either that a lock is available. or both the identity of the thread that owns the lock and a *nesting count* (described below). These locks are referred to as *nestable locks*.

The descriptions in this chapter are divided into the following topics:

- Execution environment functions (see Section 3.1, page 20).
- Lock functions (see Section 3.2, page 24).

3.1 Execution Environment Functions

The functions described in this section affect and monitor threads, and the parallel environment:

- Section 3.1.1, page 21, describes the `setNumThreads` method.
- Section 3.1.2, page 21, describes the `getNumThreads` method.
- Section 3.1.3, page 21, describes the `getMaxThreads` method.
- Section 3.1.4, page 22, describes the `getThreadNum` method.
- Section 3.1.5, page 22, describes the `inParallel` method.
- Section 3.1.6, page 22, describes the `setDynamic` method.

- Section 3.1.7, page 23, describes the `getDynamic` method.
- Section 3.1.8, page 23, describes the `setNested` method.
- Section 3.1.9, page 24, describes the `getNested` method.

3.1.1 `setNumThreads` Method

The `setNumThreads` method sets the number of threads to use for subsequent parallel regions. The format is as follows:

```
public static void setNumThreads(int numThreads)
```

The value of the parameter `numThreads` must be positive. Its effect depends whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, the value is used as the number of threads for all subsequent parallel regions prior to the next call to this method; otherwise, the value is the maximum number of threads that will be used. This method has effect only when called from a serial portions of the program. If it is called from a portion of the program, where `inParallel` returns non-zero, the behaviour of this method is undefined. For more information on this subject, see the `setDynamic` method, Section 3.1.6, page 22, and the `getDynamic` method, Section 3.1.7, page 23. This call has precedence over the `jomp.threads` system property.

> **Implementation note** *Currently, the* `jomp.runtime.OMPException` *run-time exception is thrown, when called from a portion of the program, where* `inParallel` *returns non-zero.*

3.1.2 `getNumThreads` Method

The `getNumThreads` method returns the number of threads currently in the team executing the parallel region from which it is called. The format is as follows:

```
public static int getNumThreads()
```

The `setNumThreads` method (see the preceding section) and the `jomp.threads` system property (see Section 4.2, page 27) control the number of threads in a team.

If the number of threads has not been explicitly set by the user, the default is implementation dependent. This method binds to the closest enclosing `parallel` directive (see Section 2.3, page 7). If called from a serial portion of a program, or from a nested parallel region, that is serialised, this method returns 1.

3.1.3 `getMaxThreads` Method

The `getMaxThreads` method returns the maximal value that can be returned by calls to `getNumThreads` (see Section 3.1.2, page 21). The format is as follows:

```
public static int getMaxThreads()
```

If `setNumThreads` (see Section 3.1.1, page 21) is used to change the number of threads, subsequent calls to this method will return the new value. A typical use of this method is to determine the size of an array for which all thread numbers are valid indices, even when `setDynamic` (see Section 3.1.6, page 22) is set to non-zero.

This method returns the maximum value whether executing within a serial region or a parallel region.

3.1.4 `getThreadNum` Method

The `getThreadNum` method returns the thread number, within its team, of the thread executing the method. The thread number lies between 0 and `getNumThreads()`-1, inclusive. The master thread of the team is thread 0. The format is as follows:

```
public static int getThreadNum()
```

If called from a serial region, `getThreadNum` returns 0. If called from within a nested parallel region that is serialised, this method returns 0.

3.1.5 `inParallel` Method

The `inParallel` method returns TRUE if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns FALSE . The format is as follows:

```
public static boolean inParallel()
```

This method returns TRUE from within a region executing in parallel, regardless of nested regions that are serialised.

3.1.6 `setDynamic` Method

The `setDynamic` method enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. The format is as follows:

```
public static void setDynamic(int dynamicThreads)
```

This method has effect only when called from serial portions of the program. If it is called from a portion of the program where the `inParallel` method returns non-zero, the behaviour of the method is undefined. If `dynamicThreads` evaluates to non-zero, the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the run-time environment to best utilise system resources. As a consequence, the number of threads specified by the user is the maximum thread count. The number of threads always remains fixed over the duration of each parallel region and is reported by the `getNumThreads` method (see Section 3.1.2, page 21).

If `dynamicThreads` evaluates to 0, dynamic adjustment is disabled.

A call to `setDynamic` has precedence over `jomp.dynamic` system property (see Section 4.3, page 27).

The default for the dynamic adjustment of threads is implementation dependent. As a result, user codes that depend on a specific number of threads for correct execution should explicitly disable dynamic threads. Implementations are not required to provide the ability to dynamically adjust the number of threads, but they are required to provide the interface in order to support portability across all platforms.

> **Implementation note** *Currently, the* `jomp.runtime.OMPException` *run-time exception is thrown, when called from a portion of the program, where* `inParallel` *returns non-zero.*
>
> *Current implementation does support dynamic adjustment of the number of threads.*

3.1.7 `getDynamic` Method

The `getDynamic` method returns TRUE if dynamic thread adjustment is enabled and returns FALSE otherwise. For a description of dynamic thread adjustment, see Section 3.1.6, page 22. The format is as follows:

```
public static boolean getDynamic()
```

If the implementation does not implement dynamic adjustment of the number of threads, this method always returns FALSE.

3.1.8 `setNested` Method

The `setNested` method enables or disables nested parallelism. The format is as follows:

```
public static void setNested(boolean nested)
```

If `nested` evaluates to FALSE, which is default, nested parallelism is disabled, and nested parallel regions are serialised and executed by the current thread. If `nested` evaluates to TRUE, nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form the team.

This call has precedence over the `jomp.nested` system property (see Section 4.4, page 27).

When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation dependent. As a result, implementations are allowed to serialise nested parallel regions even when nested parallelism is enabled.

> **Implementation note** *Current implementation does not support nested parallelism.*

3.1.9 `getNested` Method

The `getNested` method returns TRUE if nested parallelism is enabled and FALSE if it is disabled. For more information on nested parallelism, see the preceding section. The format is as follows:

```
public static int getNested()
```

If an implementation does not implement nested parallelism, this method always return FALSE.

3.2 Lock Functions

The methods described in this section manipulate locks used for synchronisation.

A simple lock is an object of the `jomp.runtime.Lock` class. A simple lock is created by calling a constructor with no parameters and its initial state is unlocked (no thread owns the lock). Following methods of the `jomp.runtime.Lock` class can be used to manipulate the lock.

- The `set` method waits until a simple lock is available (see Section 3.2.1, page 24).
- The `unset` method releases a simple lock (see Section 3.2.2, page 25).
- The `test` method tests a simple lock (see Section 3.2.3, page 25).

A nestable lock is an object of the `jomp.runtime.NestLock` class. A nestable lock is created by calling a constructor with no parameters and its initial state is unlocked. (no thread owns the lock). Initial nesting count is set to zero. Following methods of the `jomp.runtime.NestLock` class can be used to manipulate the lock.

- The `set` method waits until a nestable lock is available (see Section 3.2.1, page 24).
- The `unset` method releases a nestable lock (see Section 3.2.2, page 25).
- The `test` method tests a nestable lock (see Section 3.2.3, page 25).

3.2.1 `set` Method

This method blocks the thread executing the method until the lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function. The format is as follows:

```
public void set()
```

For a simple lock, ownership of the lock is granted to the thread executing the method. For a nestable lock, the nesting count is incremented, and the thread is granted, or retains, the ownership of the lock.

3.2.2 `unset` Method

This method provides the means of releasing ownership of a lock. The format is as follows:

```
public void unset()
```

The behaviour of the `unset` method is undefined if the thread executing the method does not own the lock.

For a simple lock, the `unset` method releases the thread executing the method from ownership of the lock.

For a nestable lock, the `unset` method decrements the nesting count, and releases the thread executing the method from ownership of the lock if the resulting count is zero.

3.2.3 `test` Method

This method attempts to set a lock but does not block execution of the thread. The format is as follows:

```
public boolean test()
public int test()
```

This method attempts to set a lock in the same manner as `set` does, except that it does not block execution of the thread.

For a simple lock, the `test` method returns TRUE if the lock is successfully set; otherwise, it returns FALSE.

For a nestable lock, the `test` method returns the new nesting count if the lock is successfully set; otherwise it returns zero.

Chapter 4

System Properties

This chapter describes th JOMP API system properties that control the execution of parallel code. The names of system properties must be lowercase. The values assigned to them are not case sensitive. Modifications to the values after the `jomp.runtime.OMP` class is initialised are ignored.

The system properties are as follows:

- `jomp.schedule` sets the run-time schedule type and chunk size (see Section 4.1, page 26).
- `jomp.threads` sets the number of threads to use during execution (see Section 4.2, page 27).
- `jomp.dynamic` enables or disables dynamic adjustment of the number of threads (see Section 4.3, page 27).
- `jomp.nested` enables or disables nested parallelism (see Section 4.4, page 27).

4.1 `jomp.schedule`

`jomp.schedule` applies only to `for` (see Section 2.4.1, page 8) and `parallel for` (see Section 2.5.1, page 12) directives that have schedule type `runtime`. The schedule type and chunk size for all loops can be set at a run-time by setting this system property to any of the recognised schedule types and to an optional *chunk_size*.

For `for` and `parallel for` directives that have a schedule type other than `runtime`, `jomp.schedule` is ignored. The default value for this system property is implementation dependent. If the optional *chunk_size* is set, the value must be positive. If *chunk_size* is not set, a value of 1 is assume, except in the case of the `static` schedule. For a `static` schedule, the default chunk size is set to the loop iteration space divided by the number of threads executing the loop.

Example:

```
java -Djomp.schedule=guided,4 ...
java -Djomp.schedule=static ...
```

4.2 `jomp.threads`

The value of `jomp.threads` must be positive. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, the value of this system property is the number of threads to use for each parallel region, until that number is explicitly changed during execution by calling `setNumThreads` (see Section 3.1.1, page 21).

If dynamic adjustment of the number of threads is enabled, the value of this system property is interpreted as the maximum number of threads to use.

The default value is implementation dependent.

Example:

```
java -Djomp.threads=4 ...
```

4.3 `jomp.dynamic`

The `jomp.dynamic` system property enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Its value must be TRUE or FALSE.

If set to TRUE, the number of threads that are used for executing parallel regions may be adjusted by the run-time environment to best utilise system resources.

If set to FALSE, dynamic adjustment is enabled. The default condition is implementation dependent. (For more information, see Section 3.1.6, page 22.)

Example:

```
java -Djomp.dynamic=TRUE ...
```

Implementation note *Dynamic adjustment of the number of threads is currently not implemented.*

4.4 `jomp.nested`

The `jomp.nested` system property enables or disables nested parallelism. If set to TRUE, nested parallelism is enabled; if it is set to FALSE, nested parallelism is disabled. The default value is FALSE. (For more information, see Section 3.1.8, page 23.)

Example:

```
java -Djomp.nested=TRUE
```

Implementation note *Nested parallelism is currently disabled.*

Appendix A

JOMP Grammar

Bibliography

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

[2] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.