EPCC-SS-2000-08

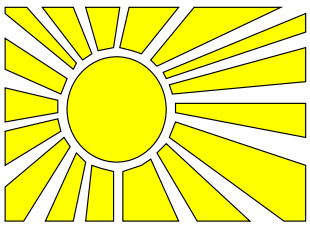
OpenMP for Java

Jan Obdržálek

Abstract

The first part of this report describes the current state of the JOMP, a definition and implementation of a set of directives and library methods for shared memory parallel programming in Java.

The rest of the paper focuses on efficiency of the current implementation. Two sets of benchmarks were implemented using JOMP, and the overheads of the synchronisation constructs as well as performance of some computationally intensive, real-world applications was measured.



Contents

1 Introduction **3**

2 The Java OpenMP API **4**
2.1 Java OpenMP Directives 4
2.1.1 `only` directive 4
2.1.2 `parallel` construct 4
2.1.3 `for` directive 4
2.1.4 `ordered` directive 5
2.1.5 `sections` and `section` directives 5
2.1.6 `single` directive 5
2.1.7 `master` directive 5
2.1.8 `critical` directive 6
2.1.9 `barrier` directive 6
2.2 Data Scope Attribute Clauses 6
2.2.1 `private` clause 6
2.2.2 `firstprivate` clause 7
2.2.3 `lastprivate` clause 7
2.2.4 `shared` clause 7
2.2.5 `default` clause 7
2.2.6 `reduction` clause 7
2.3 System Properties 8
2.3.1 `jomp.schedule` 8
2.3.2 `jomp.threads` 9
2.3.3 `jomp.dynamic` 9
2.3.4 `jomp.nested` 9
2.4 Differences from the C/C++ Standard 9
2.4.1 `threadprivate` directive and `copyin` clause 9
2.4.2 `atomic` directive 9
2.4.3 `flush` directive 9
2.4.4 Data scope attribute clauses format 9

3 The JOMP Implementation **10**
3.1 Data Clauses Support 10
3.1.1 `parallel` directive 10
3.1.2 Work-sharing directives 12
3.2 Changes to the Run-time Library 12
3.2.1 `critical` directive 12
3.2.2 `ordered` directive 14

4 OpenMP Microbenchmarks **14**
4.1 Synchronisation Overhead 14
4.1.1 Comparison with Fortran 16
4.2 Scheduling Overhead 16
4.2.1 Comparison with Fortran 16

5 Java Grande Benchmarks **18**
5.1 Section II Benchmarks 18

5.2 Section III Benchmarks 19

6 Conclusions and Future Work 20

7 Acknowledgements 21

1 Introduction

The OpenMP is a collection of compiler directives, library functions, and environment variables that can be used for shared memory parallel programming. At present, OpenMP standards exist for C/C++ [2] and for Fortran [1].

In [6], the possibility of defining and implementing OpenMP in Java, using Java's native thread model, is investigated. Possible OpenMP specification is suggested and both the JOMP compiler and runtime library are implemented.

We have continued and extended this work. First of all, and most importantly, the JOMP specification was extended to include almost all the functionality provided by OpenMP for C/C++ (see [8] for a complete specification). Both JOMP preprocessor and runtime library were updated to reflect the changes to the JOMP specification. Secondly, several bugs were fixed and some parts of the JOMP run-time library and compiler were reimplemented in a more efficient way.

The question of efficiency is very crucial for the implementation to be successful and practically usable. Therefore, two sets of benchmarks were implemented to measure the JOMP performance.

OpenMP microbenchmarks [3] were implemented in order to measure the synchronisation overheads of the JOMP directives. This information was then used to improve the JOMP run-time library performance.

Another interesting issue is how well do parallel versions of the real-world applications perform comparing to their sequential counterparts. The Java Grande Forum benchmarking suite [4] was parallelised using JOMP to get this information. Comparison with the version parallelised by hand was also made, in order to see, whether JOMP's ease to use is outweighed by a significant performance loss.

The rest of the paper is organised as follows: Section 2 briefly describes the JOMP Application Programming Interface (API), which is heavily based on the existing C/C++ specification. Section 3 describes the most important changes to the JOMP compiler and run-time library. Special attention is given to implementation of the data scope attribute clauses. In Section 4, we present some performance results for OpenMP microbenchmarks, and the results obtained using JGF benchmarking suite are to be found in Section 5. Section 6 concludes, evaluating work done, and also suggests some possible future work.

2 The Java OpenMP API

2.1 Java OpenMP Directives

In this section, a survey of available JOMP API directives and system properties is given. These are very little different from C/C++ OpenMP Standard [2] and are described in more detail in [6]. The Complete JOMP API specification can be found in [8] and a reader should refer this as the definitive reference for the JOMP API.

2.1.1 `only` directive

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

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

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

2.1.2 `parallel` construct

The `parallel` directive takes the form:

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

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

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

2.1.3 `for` directive

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

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

The loop must have a form:

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

where

<integer-type> is one of byte, short, int, long,
<relation> is one of <, <=, >=, >,
<inc> is one of <var>++, <var>-, ++<var>, -<var>, <var>+=<expr>,
<var>-=<expr>, <var>=<var>+<expr>, <var>=<var>-<expr>,
<expr> is an expression which could be implicitly cast to the same type as <var>

2.1.4 `ordered` directive

The `ordered` directive is used to specify that a block of code within the loop body must be executed for each iteration in the order that it would have been during serial execution. It takes the form:

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

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

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

2.1.6 `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] [private(<vars>)] [firstprivate(<vars>)]
  <code block>
```

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

2.1.7 `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>
```

2.1.8 `critical` directive

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

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

Only one thread may execute a critical region with a given name at any one time. Critical regions with no name specified are treated as having the same (null) name.

2.1.9 `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
```

2.2 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.

All variables, except local variables and method parameters, must have an associated type provided by a user. So the syntax of *list* is as follows:

*list ::= $TypeName_{opt}$ VariableName (, $TypeName_{opt}$ VariableName)**

2.2.1 `private` clause

SYNTAX: `private(`*list*`)`

The `private` clause declares the variables in *list* to be private to each thread in a team. A new object with automatic storage duration of the associated block is allocated. If a variable is of

class type, the default constructor with no parameters for the associated object is called. There must exist a publicly accessible constructor for this object. Declaring an array as `private` causes only a new reference to be allocated. The initial value of primitive types is indeterminate.

2.2.2 `firstprivate` clause

SYNTAX: `firstprivate(`*list*`)`

The variables specified in the `firstprivate` clause *list* have the semantics described in the `private` clause section, except that a new private object is initialised to the value of the variable's original object. A `clone()` method is used to create firstprivate instances of reference types. This method must be defined and publicly accessible for each object specified.

All variables declared to be `firstprivate` must be initialised prior to the directive on which they are declared `firstprivate`.

2.2.3 `lastprivate` clause

SYNTAX: `lastprivate(`*list*`)`

Variables specified in the `lastprivate` clause *list* have the semantics described in the `private` clause section. 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 variable's original object.

2.2.4 `shared` clause

SYNTAX: `shared(`*list*`)`

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.

2.2.5 `default` clause

SYNTAX: `default(shared|none)`

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

Specifying `default(none)` requires that each currently visible variable that is referenced in the lexical extent of parallel construct must be explicitly listed in a data scope attribute clause.

2.2.6 `reduction` clause

SYNTAX: `reduction(`*op:list*`)`
Where *op* is one of the following operators: +, $*$, -, &, ^ , |, &&, or ||.

This clause performs a reduction on the primitive type variables that appear in the *list*, with the operator *op*. A private copy of each variable in *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 re-associate the computation of the final value.

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

2.3 System Properties

The JOMP API includes several 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.

2.3.1 `jomp.schedule`

`jomp.schedule` applies only to `for` and `parallel for` directives that have schedule type `runtime` (for information on schedule types, see [8]). 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 assumed, 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 ...
```

2.3.2 `jomp.threads`

The value of `jomp.threads` must be positive. 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 the `setNumThreads` method. The default value is 1.

2.3.3 `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. Support for dynamic adjustment of the number of threads is not implemented.

2.3.4 `jomp.nested`

The `jomp.nested` system property enables or disables nested parallelism. Nested parallelism is currently not supported.

2.4 Differences from the C/C++ Standard

2.4.1 `threadprivate` directive and `copyin` clause

There is no support for the `threadprivate` directive and hence the `copyin` clause. This is because Java has no global variables as such. Problems also arise with possible orphaning of the JOMP directives.

2.4.2 `atomic` directive

This directive is not supported. The kind of optimisations which this directive is designed to facilitate (e.g. atomic updates of array elements) require access to atomic test-and-set instructions, which are not available to a Java programmer. The `critical` directive should be used instead.

2.4.3 `flush` directive

The `flush` directive is not supported, since it also requires access to special instructions. Provided that variables used for synchronisation are declared as `volatile`, this should not be a problem. However, it is not clear how the ambiguities in the Java memory model specification noted in [9] affect this issue.

2.4.4 Data scope attribute clauses format

Currently, each variable specified in the *list* of some data scope attribute clause must be given its type, unless it is a local variable or a method parameter. This is actually not a big restriction

and is included because (to keep the preprocessor as simple as possible) we do not currently have the information about all the field and package names. This would not be a issue in a full compiler, because in Java types of all the variables must be available at compile time.

3 The JOMP Implementation

3.1 Data Clauses Support

The current JOMP implementation is in some sense very primitive. The preprocessor neither performs a full semantic analysis, nor keeps a track of package, classes, variables and its names, with a single exception of local variables. It doesn't even keep a track of the current class' fields. It simply works with one compile unit at a time, and relies on a programmer to provide all necessary information. The restriction on the use of *list* in data scope attribute clauses is an immediate consequence of this fact.

To be able to handle correctly all the `private`, `firstprivate`, `lastprivate`, `shared` and `reduction` variables under these circumstances, some non-standard constructions were introduced. These are described in the following two sections together with some examples, which hopefully make the thing more clear.

In the final implementation, these problems could be solved in a much more straightforward way, assuming that we have a fully enabled compiler.

3.1.1 `parallel` directive

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

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

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

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

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

Figure 1: Hello.jomp

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

Figure 2: Hello.java

3.1.2 Work-sharing directives

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

Inside the newly allocated block, a new *Block* is created. For each variable declared to be `firstprivate, private, lastprivate,` or `reduction`, a new variable with the same name is declared. Variables declared to be `reduction` are initialised by the appropriate value. `private` and `lastprivate` variables are initialised by calling the default constructor in the case of class type variables, and uninitialised in the case of primitive or array type variables. `firstprivate` variables are initialised by the appropriate value from the `_fp_` copy of the original variable. A `clone()` method is called to initialise class or array type variables.

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

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

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

3.2 Changes to the Run-time Library

There were several changes done to the run-time library. These changes can be divided into three groups. First, the functionality of the library was extended to facilitate the full OpenMP API (e.g. reductions for more variable types and operations were added.) Second, some bugs were fixed and behaviour of few constructs was changed to be compliant with the OpenMP C/C++ standard [2]. Last, but not the least important, some constructs were re-implemented to get better performance (e.g. `critical`).

Only the major changes from the third group are described in this section. There were many other changes done, but the run-time library description in [6] still applies without any amendments.

3.2.1 `critical` directive

Nested locks are no longer used to implement the `critical` directive. Instead of this, the structured block associated with the directive is enclosed in a *synchronized* statement. Locks

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

Figure 3: a fragment of the JOMP program

```
{ // OMP FOR BLOCK BEGINS
  // copy of firstprivate variables, initialised
  int _cp_i = i;
  // copy of lastprivate variables
  int _cp_k;
  // variables to hold result of reduction
  int _cp_l;
  boolean amLast=false;
  { // Inner loop
    // firstprivate variables + init
    int  i = (int) _cp_i;
    // [last]private variables
    int j;
    int k;
    // reduction variables + init to default
    int  l = 0;

     ... code to handle the parallel loop ...

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

Figure 4: Resulting java code

passed as a parameter are held in a static hash table and the `getLockByName` method is used to get a reference to the lock associated with a given name, creating it if necessary.

3.2.2 `ordered` directive

The implementation of the `Orderer` class was changed to get better performance. There are two new arrays: `Locks[]` and `Iters[]`, with one element per every thread in a team. Every `Iters[`*i*`]` variable holds a number of the next iteration to be performed by the thread *i*. This information is used in order to notify only the thread which is to perform the sequentially next iteration. The `Locks[]` array is used to synchronise the access to the `Iters[]` array elements.

4 OpenMP Microbenchmarks

The cost of synchronisation overhead is crucial for overall performance of the JOMP applications. To get this information, a set of OpenMP microbenchmarks was parallelised using JOMP directives. The methodology we used is the one described in [3] and consists of comparing the time needed to execute the same code with and without each directive. Comparison with the results obtained for Fortran is also shown.

All the benchmarks were run on Sun HPC 6500 system with 18 400MHz UltraSPARCII processors, each having 8MB of second level cache, and with 18GB of shared memory. The JVM used was Sun's Solaris production JDK, version 1.2.1_04. Options `-Xmx512m` and `-Xms10m` were used. Fortran version of the benchmarks was compiled using the KAI guidef90 compiler, version 3.7 and then the Sun WorkShop 5.0 f90 compiler, with options `-fast -xarch=v8plusa`.

4.1 Synchronisation Overhead

On figure 5, synchronisation overhead for those directives, which use only the internal barrier for synchronisation, is shown. The actual scalability is pretty good and these directives are probably not amenable to much further optimisation.

Figure 6 shows the synchronisation overhead for directives using Java's *synchronized* statement. As expected, the `critical` directive and the `lock/unlock` pair of locking methods have almost the same performance. However, the *synchronized* statement (at least in Sun's JDK) is quite expensive. The performance of `ordered` is improved using some optimisations mentioned earlier.

The bad results for the `single` directive (see table 1), which is not shown in this graph, are because of some odd behaviour of the benchmarks and also because of not the implementation of the Java's *synchronized* statement. However, in real-world applications, the overhead is expected to be much smaller. On the other hand, the `single` directive with a `nowait` clause specified performs really well. It would be interesting to compare this to the results obtained using Java Virtual Machines other than the one from Sun's JDK.

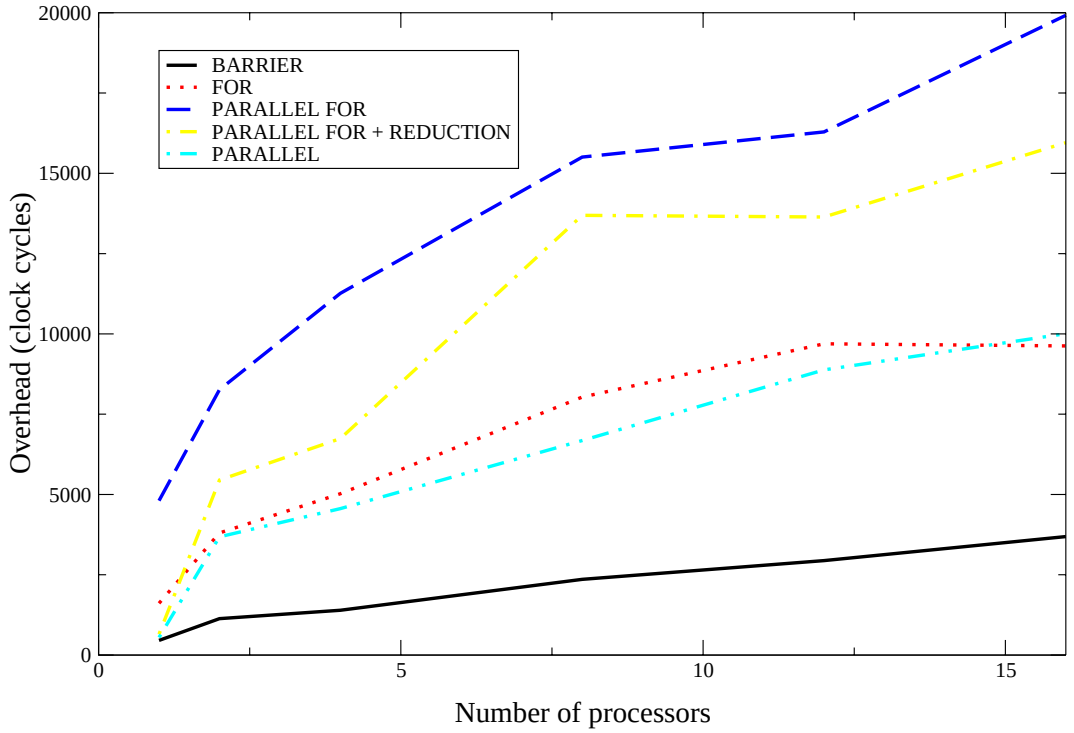


Figure 5: Synchronisation overhead on Sun HPC 6500

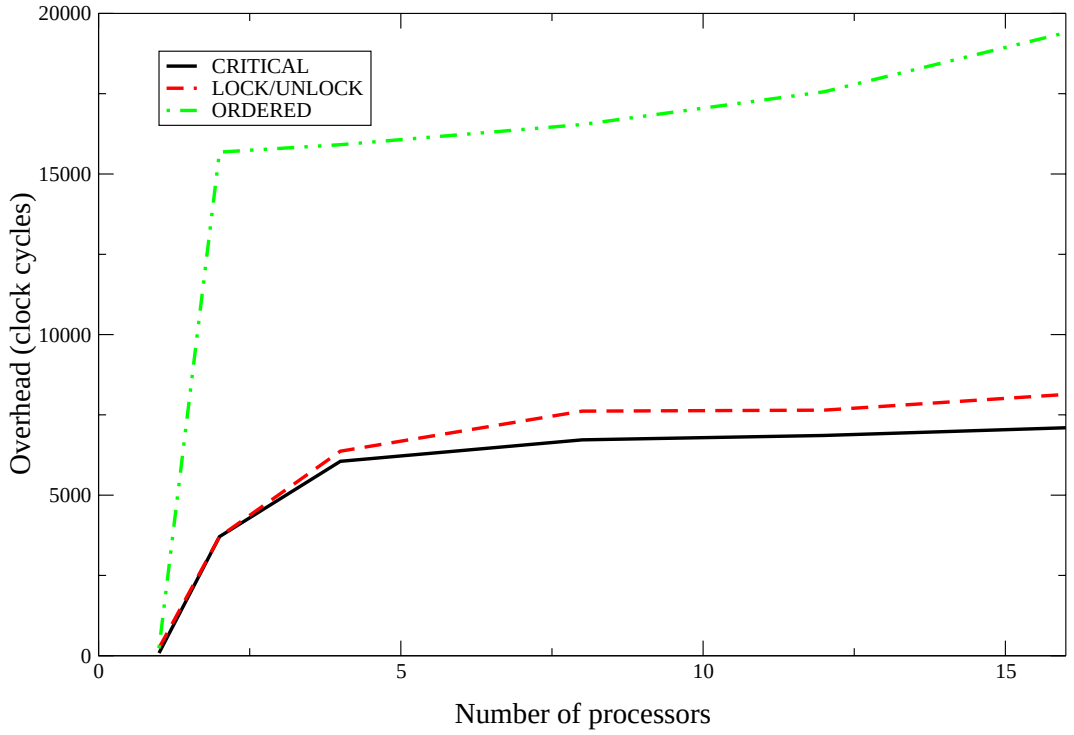


Figure 6: Synchronisation overhead on Sun HPC 6500

4.1.1 Comparison with Fortran

Table 1 shows the overhead of various constructs obtained both for the JOMP and for guidef90 Fortran compiler. Without a single exception, all the JOMP directives requiring only barrier synchronisation outperform their guidef90 Fortran counterparts, as the basic routine is almost 4 times faster.

The overheads of the locking-type synchronisation are somewhat higher in JOMP, however still small enough. In JOMP, the overhead of these directives is determined by the cost of Java's *synchronized* statements and probably can not be significantly decreased.

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

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

4.2 Scheduling Overhead

This section of the OpenMP microbenchmarks deals with another interesting question: how big is the synchronisation overhead for different scheduling strategies used by the JOMP `for` directive? Figure 4.2 show the results for 8 processors. Note, that a logarithmic scale is used to show the overhead.

As can be expected, performance of the `dynamic` and `guided` schedules is much worse then for the `static` schedule, clearly because of the cost of Java's *synchronized* construct. This is reduced with an increasing *chunk_size*, which also agrees with what we would expect. The behaviour of the `static` schedule with a given *chunk_size* is quite interesting. It actually faster then the default `static` schedule and the difference grows with the *chunk_size*. It would be interesting to find out why this happens.

4.2.1 Comparison with Fortran

Table 2 shows the overhead of `for` using different schedules, both for the JOMP and for guidef90 Fortran compiler. Except for the `static` schedule, results are poor because of using the *synchronized* statement.

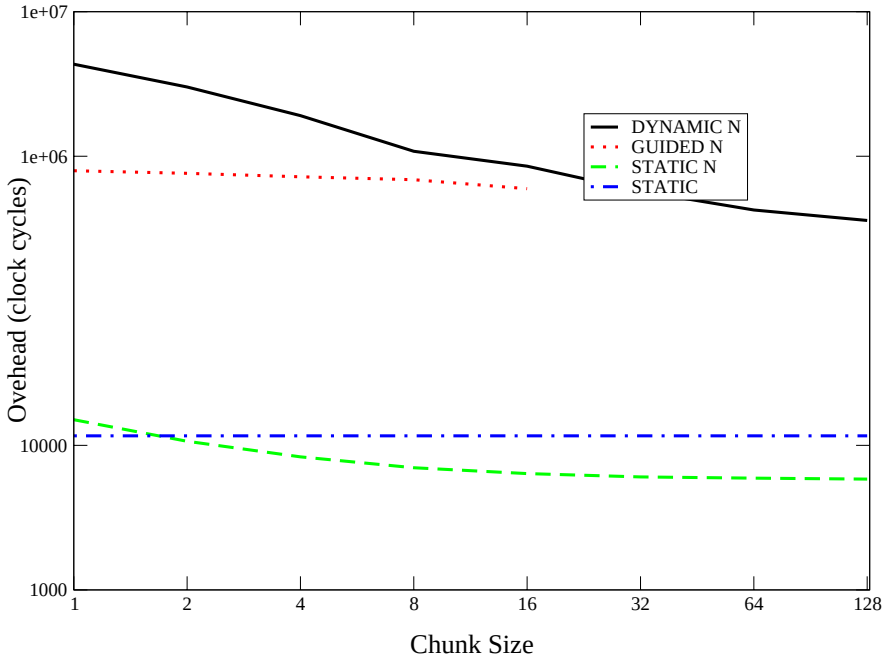


Figure 7: Scheduling overhead on Sun HPC 6500

Schedule	size	guidef90	JOMP
STATIC	-	11.98	29.08
	1	18.93	37.62
	2	15.48	26.68
	4	15.75	20.82
	8	12.22	17.50
	16	14.54	15.93
	32	11.41	15.12
	64	13.94	14.82
	128	10.38	14.59
DYNAMIC	1	379.98	10834.24
	2	204.50	7530.37
	4	91.08	4776.36
	8	48.30	2706.15
	16	36.86	2135.08
	32	31.55	1432.39
	64	26.90	1062.52
	128	28.64	898.88
GUIDED	1	293.27	1985.00
	2	228.41	1904.26
	4	182.95	1802.84
	8	141.30	1719.89
	16	101.96	1490.71

Table 2: Scheduling overhead (in microseconds) on Sun HPC 6500

5 Java Grande Benchmarks

A *Grande Application* is defined [5] as an application of large-scale nature, potentially requiring large amounts of processing power, network bandwidth, I/O, and memory. *Grande* applications often rely on parallel execution environments to provide the necessary computational power.

Java has the greatest potential to deliver an attractive productive programming environment spanning the very broad range of tasks needed by a Grande programmer - it is portable, object oriented, easy to use and has built-in support for parallelism.

Therefore Java Grande applications are a natural choice for being parallelised in order to get better application performance. Java Grande Benchmarks (see [4]) were parallelised using JOMP and performance tests were run. Speedup graphs are shown and the results are compared to the ones obtained by parallelising benchmarks by hand [7] (we refer to this manually parallelised version as to the *threaded* version).

To interpret the results correctly, we must bear in mind two things. First, parallelising programs using JOMP is usually much faster and more straightforward. Second, the threaded version usually supplies every thread with its own copy of all variables used in the computation. This allows the threaded version to be occasionally faster then its JOMP counterpart. However, the JOMP version is much more readable and easier to understand.

Speedup graphs for all the JGF benchmarks are included in Appendix A.

5.1 Section II Benchmarks

The kernel benchmarks are chosen to be short codes, executed in parallel (multi-threaded), each containing a type of computation likely to be found in Grande applications.

Series Series computes the first N Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval (0,2). Performance units are coefficients per second. This benchmark heavily uses transcendental and trigonometric functions.

Main loop was parallelised using the `parallel for` directive.

LU Factorisation LU Factorisation solves an NxN linear system using LU factorisation followed by a triangular solve. This is a Java version of the well known Linpack benchmark. Performance units are Mflops per second. This benchmark is memory and floating point intensive.

The Gaussian elimination is the only part of the computation which is decomposed, for the rest of the code is inherently serial. The `parallel for` directive was used to parallelise this loop. The speedup is not linear, but still scales pretty well.

Crypt Crypt performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. Performance units are bytes per second. Bit/byte operation intensive.

Main encrypting/decrypting loop was parallelised using the `parallel for` directive. Nice linear speedup.

SOR The SOR benchmark performs 100 iterations of successive over-relaxation on a NxN grid. The performance reported is in iterations per second.

Main loop was parallelised using a single `parallel for` directive. There were some changes to the code in both the threaded and JOMP versions to permit parallelisation. The super-linear speedup is probably due to cache effects.

Sparse This benchmark uses an unstructured sparse matrix stored in compressed-row format with a prescribed structure. This kernel exercises indirection addressing and non-regular memory references. A N x N sparse matrix is used for 200 iterations.

The main loop is parallelised using the `parallel` directive. Limits of this loop are set individually for each thread, using the `getThreadNum()` method.

5.2 Section III Benchmarks

The section III benchmarks are application benchmarks intended to be representative of Grande applications, suitably modified for inclusion in the benchmark suite by removing any I/O and graphical components. The same approach as in the section II benchmarks has been taken into decomposing the existing code.

Monte Carlo This benchmark is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates N sample time series with the same mean and fluctuation as a series of historical data. Performance is measured in samples per second.

The main loop was parallelised using the `parallel for` directive and results are combined in a `critical` section.

Ray Tracer This benchmark measures the performance of a 3D ray tracer. The scene rendered contains 64 spheres, and is rendered at a resolution of NxN pixels. The performance is measured in pixels per second. Main loop was parallelised using the `parallel for` directive. Every thread is assigned a part of the scene to be rendered.

The poor performance of the JOMP version is because of not implementing some optimisations, which are present in the sequential version. "Global" variables, which we cannot make private to a thread, are used there to speed up the computation. In the threaded version every thread has its own copy of the scene and environment, so it still can profit from the sequential optimisations.

Euler The Euler benchmark solves the time-dependent Euler equations for flow in a channel with a "bump" on one of the walls. A structured, irregular, N x 4N mesh is employed, and the solution method is a finite volume scheme using a fourth order Runge-Kutta method with both second and fourth order damping. The solution is iterated for 200 timesteps. Performance is reported in units of timesteps per second.

The code of the main method was enclosed in a `parallel` region. This method contains several *for* loops (14), of which every one was parallelised using the `for` directive, and the remaining code was serialised using the `master` directive. Actual speedup is far from being linear.

6 Conclusions and Future Work

The current state of the JOMP compiler and run-time library has been presented, and some design issues discussed. By now, JOMP implements almost all the functionality of C/C++ OpenMP API. Benchmarks to measure the synchronisation overhead of low-level constructs have been implemented and the overhead has been measured and found to be quite reasonable. A further analysis shows that the efficiency these constructs depends much on the efficiency of Java's *synchronized* statement.

To measure the performance of some real world applications, JGF benchmarks have been implemented, and performance tests have been run. The resulting code scales well, with little overhead comparing to a hand-coded Java threads version. This proves that OpenMP-like approach is suitable and useful for parallelising Java programs.

Further work includes a complete specification, draft of which can be found in [8]. In the rest of this section, some other outstanding issues are outlined.

Fully enabled preprocessor A preprocessor performing full semantic analysis (or at least full analysis of *names*) is probably the next step in JOMP evolution. Having information about all the package/class/method names and about binding identifiers to particular variables would made the JOMP preprocessor in some sense more straightforward to implement - especially its treatment of data scope attribute clauses. As the source code for Sun's JDK javac compiler is available, this could be done in reasonable way, since all the complex naming conventions are already implemented.

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. This issue could possibly be addressed by the full-flagged preprocessor mentioned earlier.

Directive format In Java, line terminators are no more significant for the syntactic analysis of code than other white space characters. As a consequence of this, a JOMP directive can have its clauses on multiple lines, which are not syntactically required to begin with the `//omp` sentinel. This breaks the compatibility with sequential compilers, just because such a directive is not treated as a one line comment.

A possible solution to this problem is to use multiple line comments to designate JOMP directives. Then the JOMP sentinel would be `/*omp` and a `omp*/` terminator has to be inserted after the last clause. This construct should be recognised equally well by both JOMP-aware and sequential compilers.

Array reductions As in C/C++ OpenMP [2], only reductions for scalar variables are implemented. It would be interesting to extend the semantics of reduction to (at least) onedimensional arrays of primitive types. This should be not too hard to implement, because the length (and type) of the array is known at run time.

7 Acknowledgements

First of all, I would like to thank my supervisor, Mark Bull, for continuous support, guidance and encouragement. Thanks are also due to Lorna Smith (not only) for her help with the JGF benchmarks and to my fellow SSP students.

Jan Obdržálek is partially supported by the Grant Agency of the Czech Republic, grants No. 201/00/0400 and No. 201/00/1023.

References

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

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

[3] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, pages 99–105, Sept. 1999.

[4] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 81–88, July 1999.

[5] JGF. Making Java work for high-end computing. Technical Report JGF-TR-1, Java Grande Forum, 1998.

[6] Mark Kambites. Java OpenMP. Technical Report EPCC-SS99-05, Edinburgh Parallel Computing Centre, 1999.

[7] Alexandros Karatzoglou. Developing a parallel benchmarking suite for Java Grande applications. Technical Report EPCC-SS99-06, Edinburgh Parallel Computing Centre, 1999.

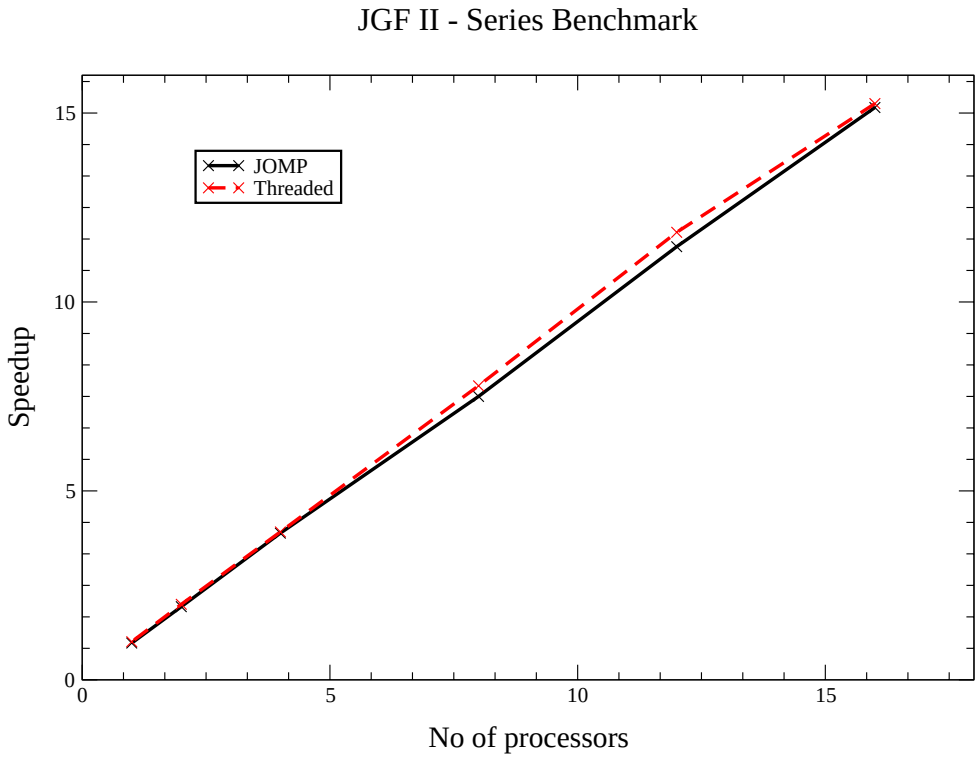
[8] Jan Obdržálek and Mark Bull. JOMP Application Program Interface, version 0.1 (draft). Technical report, Edinburgh Parallel Computing Centre, 2000. Available from `www.epcc.ed.ac.uk/research/jomp`.

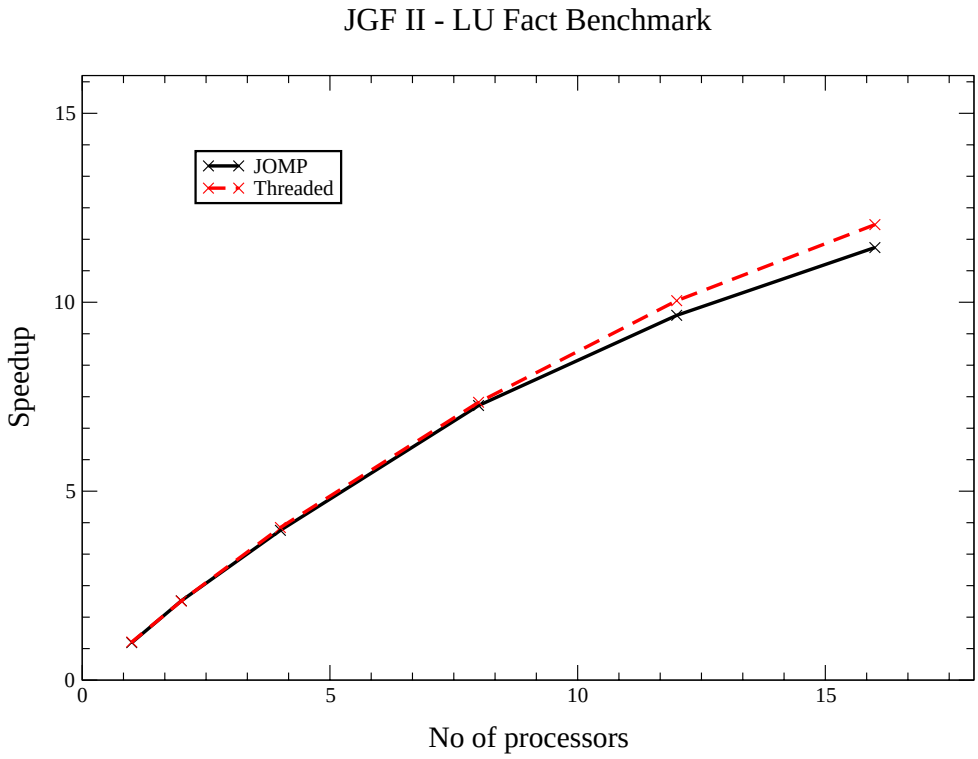
[9] W. Pugh. Fixing the Java memory model. In *Proceedings of the ACM Java Grande Conference*, pages 89–98, June 1999.

Jan Obdržálek is currently studying for Masters degree in Computer Science at Faculty of Informatics, Masaryk University, Brno, Czech Republic. He is interested in concurrency theory - particularly in modal and temporal logics of processes and model checking, formal verification and process algebras.

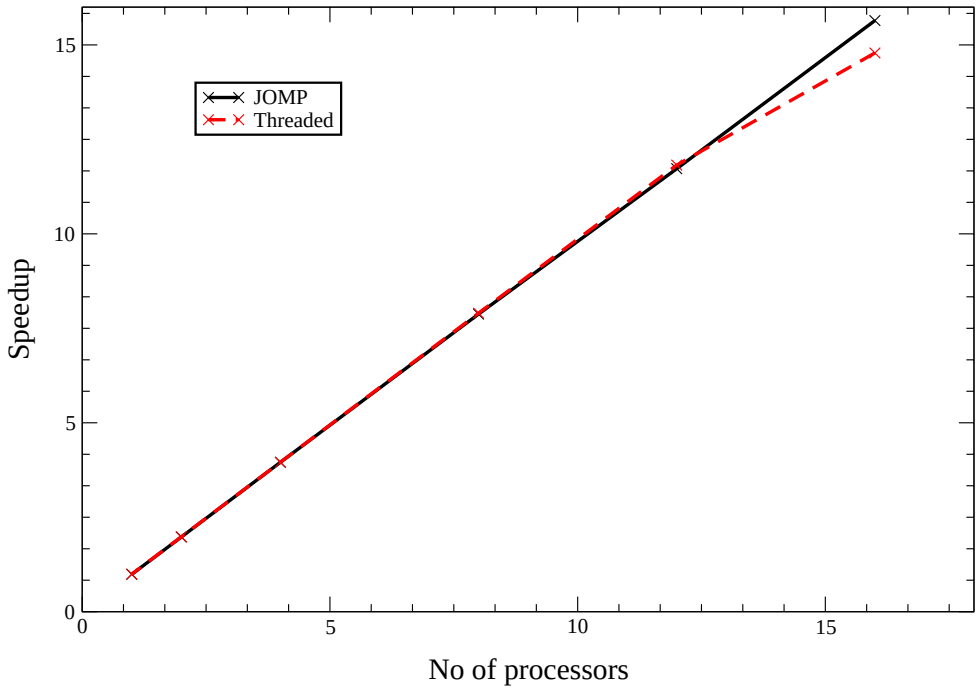
This project was supervised by Dr. Mark Bull.

Appendix A - JGF Benchmarks - Results

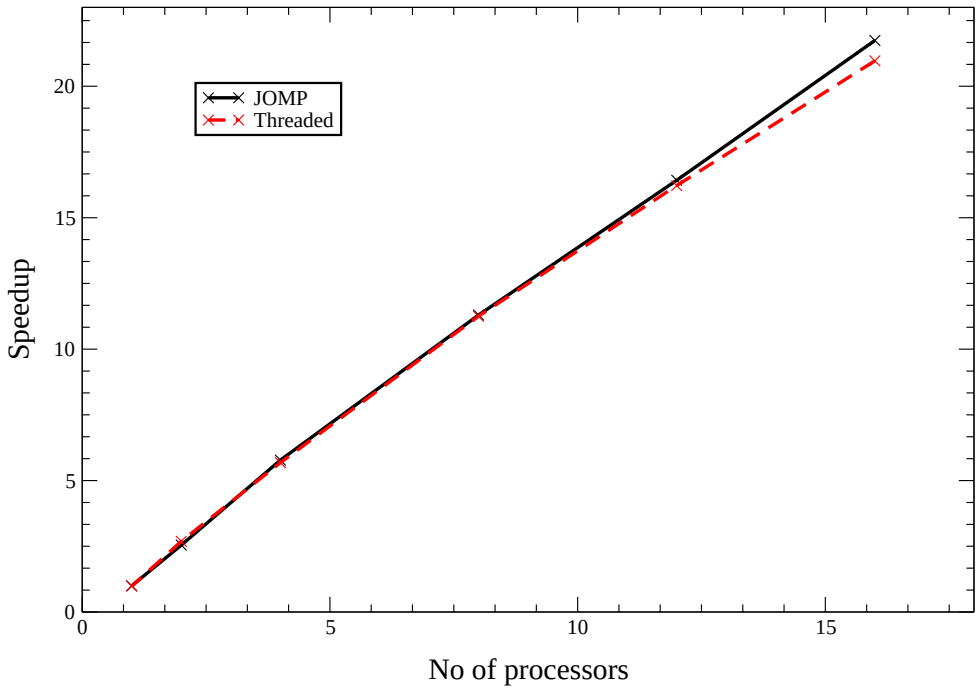




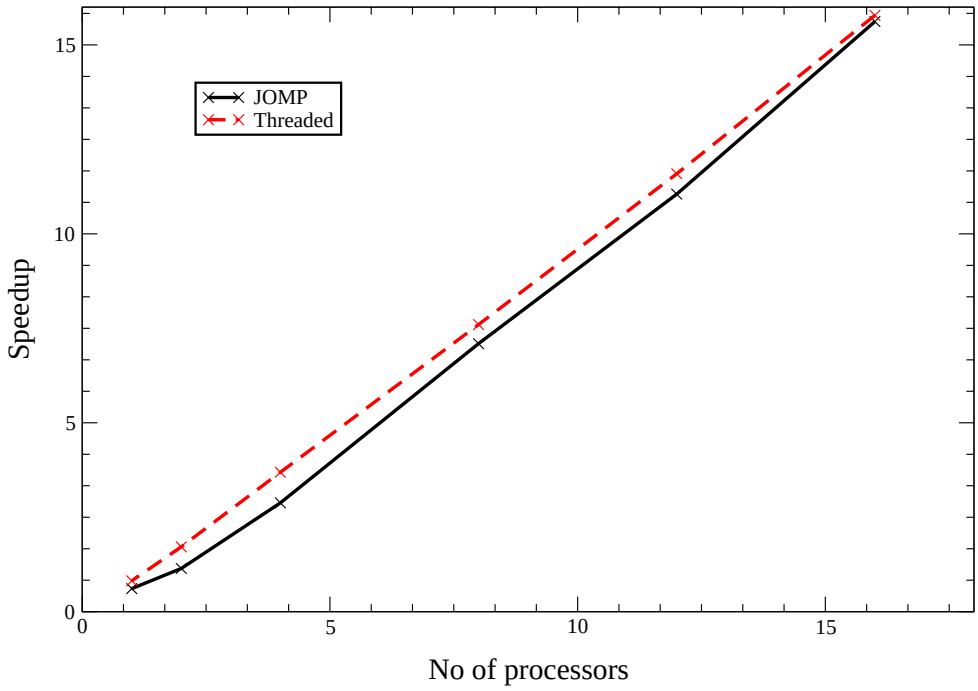
JGF II - Crypt Benchmark



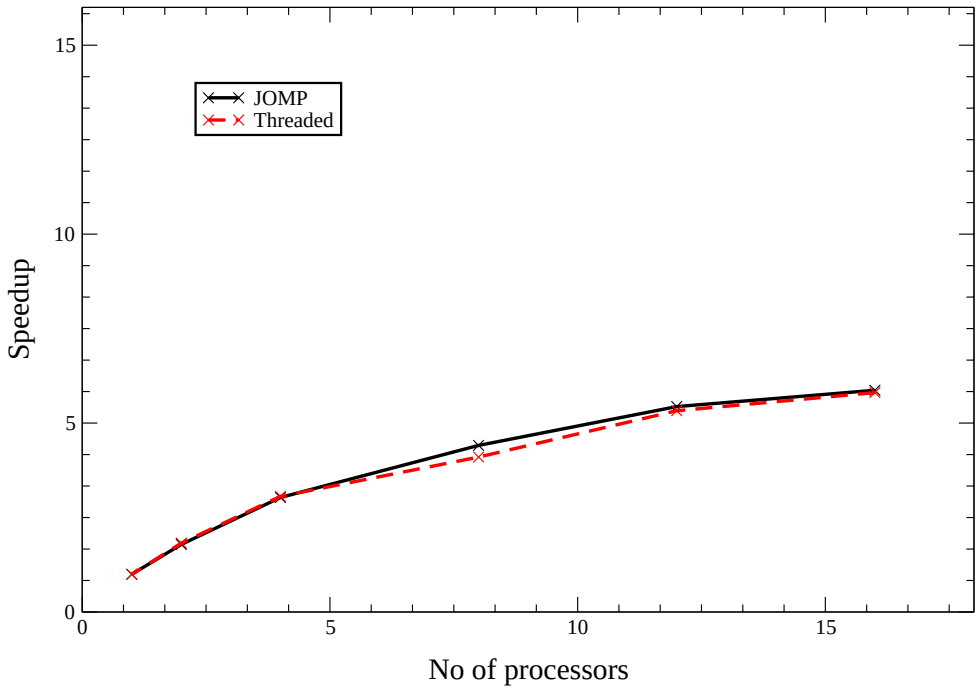
JGF II - SOR Benchmark



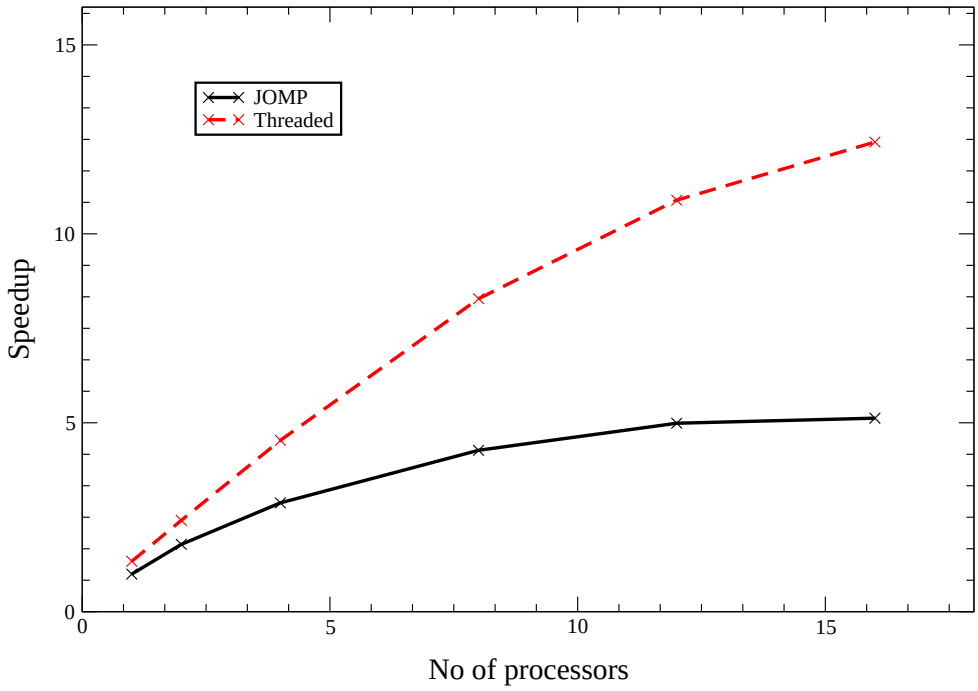
JGF II - Sparse Benchmark



JGF III - Monte Carlo Benchmark



JGF III - Ray Tracer Benchmark



JGF III - Euler Benchmark