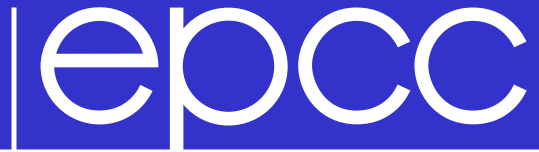


JOMP
An OpenMP-like interface for Java

Dr. J. Mark Bull
EPCC, University of Edinburgh, UK
m.bull@epcc.ed.ac.uk

Mark Kambites
Dept. of Mathematics, University of York, UK

- Relatively new standard for shared memory parallel programming.

- Has wide vendor support, through the OpenMP Architecture Review Board.

- Defines sets of compiler directives, library routines and environment variables both for Fortran 77/90 and for C/C++.

- Higher level of abstraction than, say, POSIX threads.

- Thread class part of standard Java libraries.
- Most current JVMs implement Java threads on top of native system threads.
- Cannot afford to create a new thread for each task.

- Need to implement a task pool
  - one thread per processor
  - threads busy wait when idle for short periods.
- Also need:
  - fast barrier synchronisation for SPMD programs.
  - loop scheduling algorithms.

Why OpenMP for Java?

- Implementing e.g. parallel loops using Java threads is a bit messy.
  - Need to define new class with a method containing the loop body and pass an instance of this to the task pool.
  - Code changes become harder to implement.
- Relatively simple to automate the process using compiler directives.
- OpenMP is becoming increasingly familiar to Fortran and C/C++ programmers in HPC.
- Using directives allows easy maintenance of a single version of source code.

An OpenMP-like API for Java

- Based heavily on the C/C++ OpenMP standard.
- Directives embedded as comments (as in Fortran)

```
//omp <directive> <clauses>
```

- Library functions are class methods of an OMP class
- Java system properties take place of environment variables.

API

- Principal construct is PARALLEL REGION (all threads execute enclosed code)
- Most other OpenMP directives supported:
  - FOR (parallel for loops with scheduling options)
  - SECTIONS (parallel independent code blocks)
  - CRITICAL (mutual exclusion)
  - SINGLE (one thread only)
  - MASTER (master thread only)
  - BARRIER (explicit barrier synchronisation)
  - ONLY (conditional compilation)
- Data attribute scoping (SHARED, PRIVATE REDUCTION clauses)

API (cont.)

- Library routines:
  - get and set # of threads.
  - get thread id.
  - determine whether in parallel region
  - enable/disable nested parallelism
  - simple and nested locks
- System properties:
  - set # of threads
  - set loop scheduling options
  - enable/disable nested parallelism

API (cont.)

- Some differences from C/C++
  - no ATOMIC directive
  - no FLUSH directive
  - no THREADPRIVATE directive

- Outstanding issues
  - restrictions on scope of variables declared in data attribute clauses.
  - exception handling
  - task based parallelism (while loops, recursive calls)

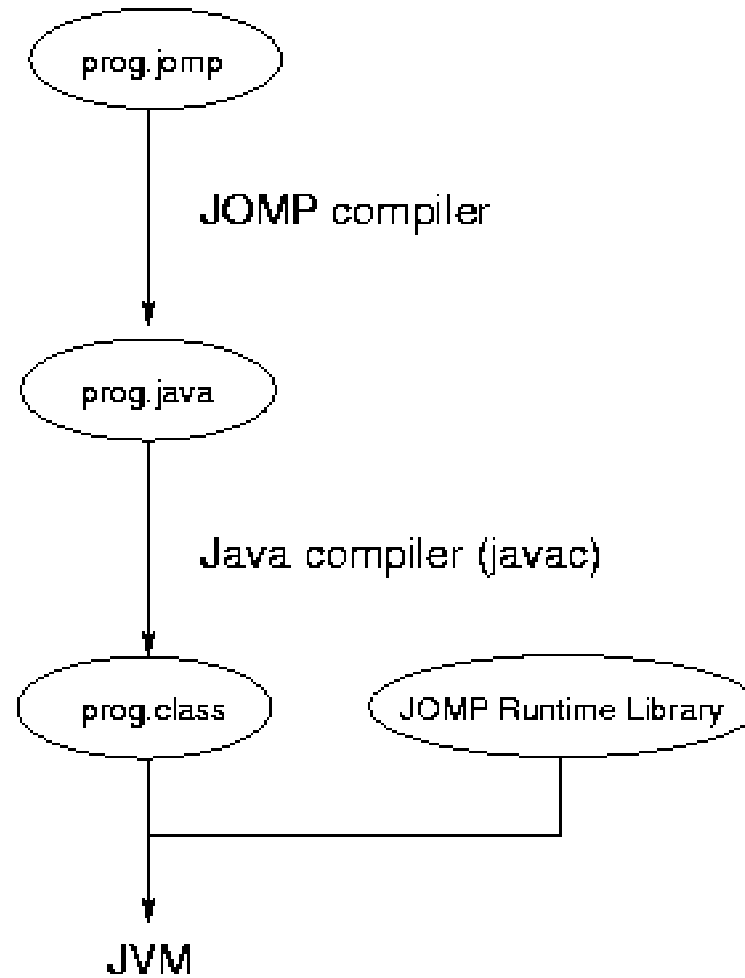
Example

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

JOMP compiler

- Built using JavaCC, and based on the free Java 1.1 grammar distributed with JavaCC.
- JOMP is written in Java, so is fully portable!
- Java source code is parsed to produce an abstract syntax tree and symbol table.
- Directives are added to the grammar.
- To implement them, JOMP overrides methods in the unparsing phase.
- Output is pure Java with calls to runtime library.

JOMP system



Implementing a parallel region

- On encountering a parallel region, the compiler creates a new inner class.
- The inner class has a `go` method, containing the code inside the region, and declarations of private variables.
- The inner class contains data members corresponding to shared and reduction variables.
  - need to take care with initialisation (Java compilers are somewhat pedantic!)
  - more copying required than in C, (no `varargs` equivalent)
- A new instance of the inner class is created, and passed to the runtime library, which causes the `go` method to be executed on each thread.

Parallel “Hello World”

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

"Hello World" implementation

```
import jomp.runtime.*;

public class Hello {
   public static void main (String argv[]) {
      int myid;
      __omp_class_0 __omp_obj_0 = new __omp_class_0();
      try {
         jomp.runtime.OMP.doParallel(__omp_obj_0);
      }
      catch (Throwable __omp_exception) {
         jomp.runtime.OMP.errorMessage();
      }
   }
```

epcc

```
  private static class __omp_class_0
          extends jomp.runtime.BusyTask {
      public void go (int __omp_me) throws Throwable {
          int myid;
          myid = OMP.grtThreadNum();
          System.out.println(“Hello from “ + myid);
      }
  }
}
```

Runtime library

- Performs thread management and assigns tasks to be run to the threads.
- Implements fast barrier synchronisation (lock-free F-way tournament algorithm).
- Uses a variant of the barrier code to implement fast reductions.
- Support for static and dynamic loop scheduling, and ordered sections in a loop.
- Implements locks and critical regions using `synchronized` blocks.

Example application

- Monte Carlo financial simulation (from JGF benchmark suite)
- Coarse grained loop parallelism.
- 6.6 speedup on 8 processor Sun E3500.
- Very similar performance to hand coded Java threads version.

Future work

- Produce full language specification.
- Fill in some missing functionality.
- Make publicly available for testing and comments.
- Performance analysis and tuning.
- More applications!