Towards OpenMP for Java

Mark Bull and Martin Westhead
EPCC, University of Edinburgh, UK

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

Jan Obdrzalek
Masaryk University, Brno, Czech Rebublic

Overview

- Java, Java threads and Java for HPC
- JOMP API specification
- JOMP compiler
- JOMP runtime library
- Performance

- How Java works:
  - Source code is compiled to platform independent byte code
  - Byte code is executed by the Java virtual machine (JVM)
  - JVMs exist for almost every platform
- Strongly typed and O-O language:
  - No global variables
  - Objects are manipulated using references
  - No accessible pointers
  - Automatic garbage collection

- Advantages
  - Portability (even at binary level)
  - Easy and rapid development
  - Built-in security
  - Built-in parallelism support
- Problems
  - performance (though latest JVMs are not too bad)
  - no complex numbers, no multidimensional arrays
  - lack of high-level parallel programming standards

Java threads

- Thread class part of standard Java libraries.
- Threads are objects (instances of thread class).
- Fork-join execution model.
- The **`start()`** method is called to start a thread
- Each thread then executes its **`run()`** method.
- Synchronisation is via monitors associated with objects
- **`synchronized`** methods and blocks

- 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

- Most OpenMP directives supported:
  - PARALLEL
  - FOR
  - SECTIONS
  - CRITICAL
  - SINGLE
  - MASTER
  - BARRIER
  - ONLY (conditional compilation)
- Data attribute scoping
  - DEFAULT, SHARED, PRIVATE FIRSTPRIVATE, LASTPRIVATE and REDUCTION clauses

epcc

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 (`java -Djomp.threads=8 MyProg`)
  - set loop scheduling options
  - enable/disable nested parallelism

API (cont.)

- Some differences from C/C++ API:
  - no ATOMIC directive
  - no FLUSH directive
  - no THREADPRIVATE directive
  - REDUCTION for arrays (not implemented yet)
  - no function to return number of processors

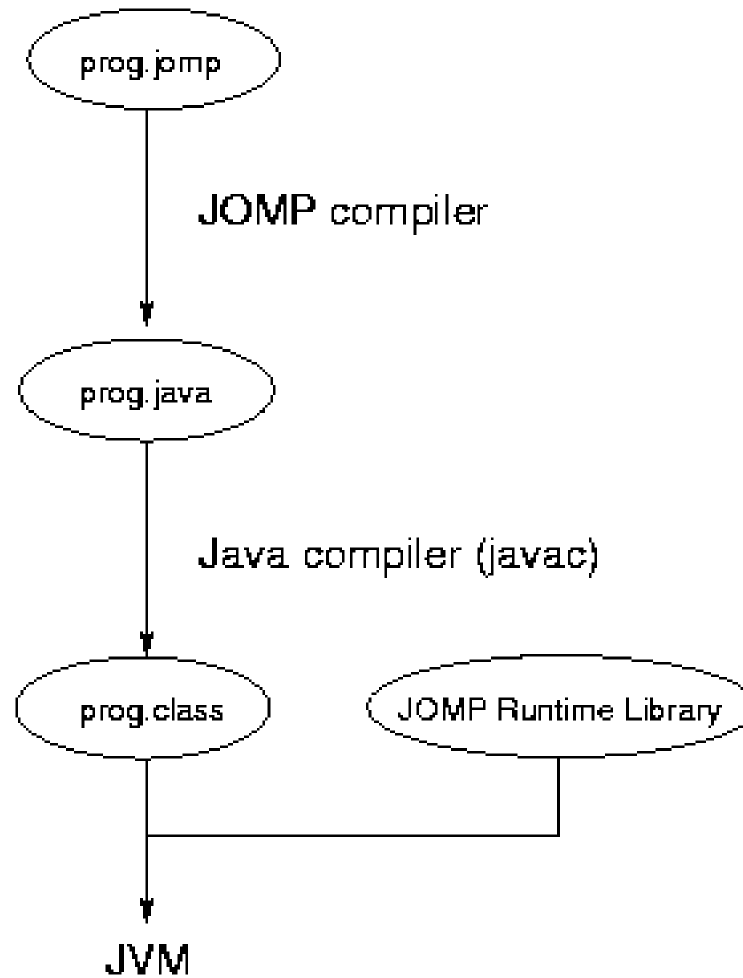
Example

```
//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 class.
- The class has a **`go()`** method, containing the code inside the region, and declarations of private variables.
- The 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 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();
      }
   }
```

```
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);
    }
  }
}
```

Implementation (cont.)

- By simulating the original name scope, original code block is reused verbatim.
- Worksharing directives are replaced with additional code for e.g. loop scheduling
- Local variables used to simulate original name scope
- Use an inner class for DEFAULT(SHARED), a normal class for DEFAULT(NONE).

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.

Performance

- Java version of EPCC OpenMP microbenchmarks
- Measures overhead of directives
- Uses native clock routines
- Results are from Sun E6500, 16 (out of 18) 400MHz processors, Sun JDK 1.2.1_04

Microbenchmark results

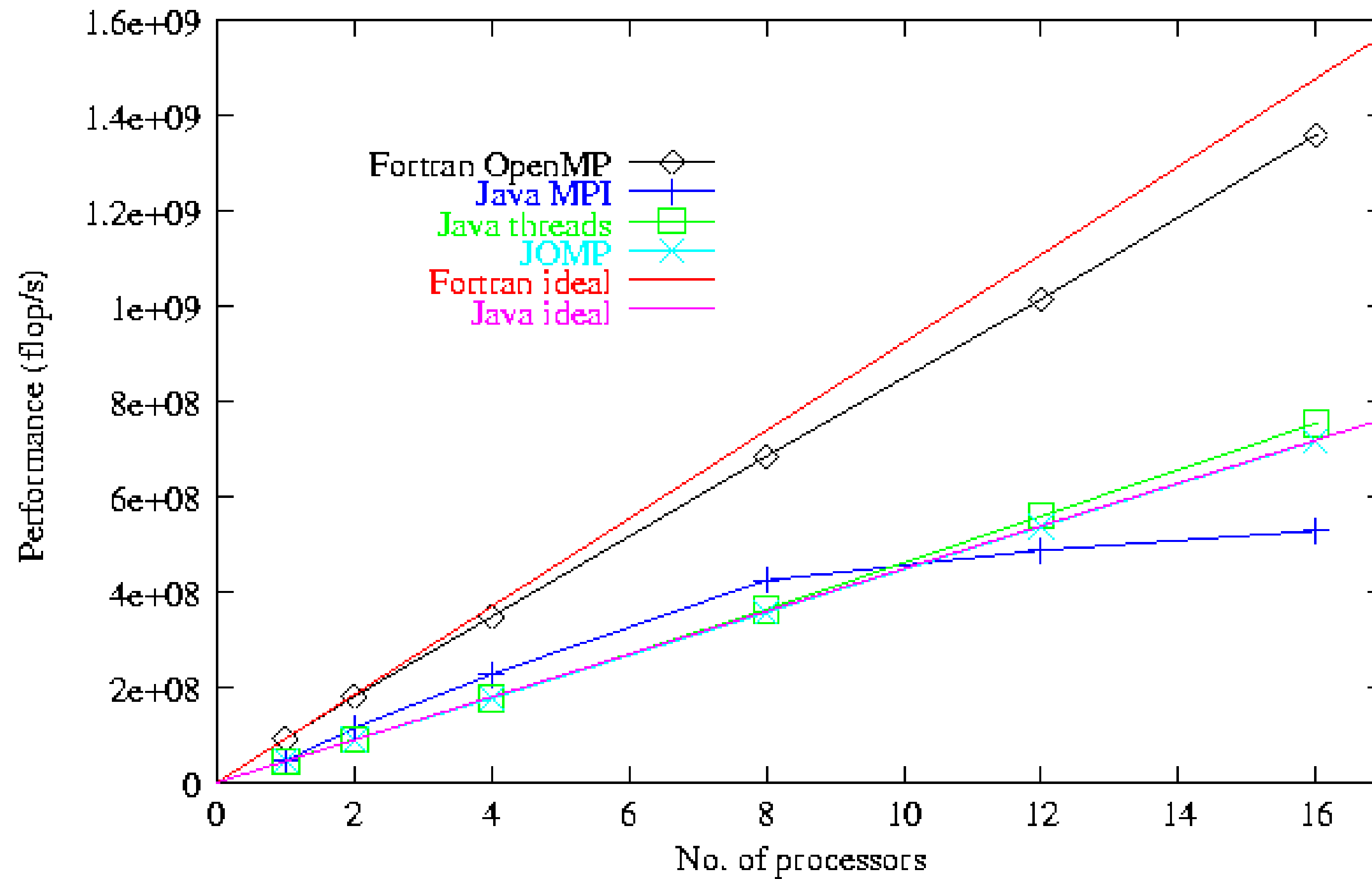
Directive	guidef90	JOMP
PARALLEL	78.4	34.3
PARALLEL + REDUCTION	166.5	58.4
BARRIER	41.7	11.0
DO/FOR	42.3	24.6
SINGLE	83.0	1293
CRITICAL	11.2	19.1

- JOMP is competitive on BARRIER synchronisation
- Limited by performance of Java's **`synchronized`** construct

CFD kernel benchmark

- Simple 2-D , steady-state viscous fluid flow in a box
- Red-black Gauss-Seidel on regular grid, 5-point stencil
- Parallelised with two PARALLEL FOR directives
- Compared to Fortran 90 + OpenMP (KAI guidef90), hand coded Java threads, and a Java binding to MPICH.

CFD kernel benchmark



Future work

- Refine API specification.
- Fill in missing functionality.
- Performance portability studies
- More applications!

JOMP homepage

www.epcc.ed.ac.uk/research/jomp

- Beta version of JOMP compiler and runtime library
- First draft of API specification
- JOMP microbenchmarks
- JOMP versions of Java Grande Forum benchmarks